Linux Driver-api Documentation

The kernel development community

Jul 14, 2020

CONTENTS

The kernel offers a wide variety of interfaces to support the development of device drivers. This document is an only somewhat organized collection of some of those interfaces —it will hopefully get better over time! The available subsections can be seen below.

Table of contents

CHAPTER

ONE

DRIVER MODEL

1.1 Driver Binding

Driver binding is the process of associating a device with a device driver that can control it. Bus drivers have typically handled this because there have been bus-specific structures to represent the devices and the drivers. With generic device and device driver structures, most of the binding can take place using common code.

1.1.1 Bus

The bus type structure contains a list of all devices that are on that bus type in the system. When device_register is called for a device, it is inserted into the end of this list. The bus object also contains a list of all drivers of that bus type. When driver_register is called for a driver, it is inserted at the end of this list. These are the two events which trigger driver binding.

1.1.2 device_register

When a new device is added, the bus' s list of drivers is iterated over to find one that supports it. In order to determine that, the device ID of the device must match one of the device IDs that the driver supports. The format and semantics for comparing IDs is bus-specific. Instead of trying to derive a complex state machine and matching algorithm, it is up to the bus driver to provide a callback to compare a device against the IDs of a driver. The bus returns 1 if a match was found; 0 otherwise.

int match(struct device * dev, struct device_driver * drv);

If a match is found, the device' s driver field is set to the driver and the driver' s probe callback is called. This gives the driver a chance to verify that it really does support the hardware, and that it' s in a working state.

1.1.3 Device Class

Upon the successful completion of probe, the device is registered with the class to which it belongs. Device drivers belong to one and only one class, and that is set in the driver' s devclass field. devclass_add_device is called to enumerate the device within the class and actually register it with the class, which happens with the class' s register_dev callback.

1.1.4 Driver

When a driver is attached to a device, the device is inserted into the driver' s list of devices.

1.1.5 sysfs

A symlink is created in the bus' s 'devices' directory that points to the device' s directory in the physical hierarchy.

A symlink is created in the driver' s 'devices' directory that points to the device' s directory in the physical hierarchy.

A directory for the device is created in the class' s directory. A symlink is created in that directory that points to the device' s physical location in the sysfs tree.

A symlink can be created (though this isn' t done yet) in the device' s physical directory to either its class directory, or the class' s top-level directory. One can also be created to point to its driver' s directory also.

1.1.6 driver_register

The process is almost identical for when a new driver is added. The bus' s list of devices is iterated over to find a match. Devices that already have a driver are skipped. All the devices are iterated over, to bind as many devices as possible to the driver.

1.1.7 Removal

When a device is removed, the reference count for it will eventually go to 0. When it does, the remove callback of the driver is called. It is removed from the driver' s list of devices and the reference count of the driver is decremented. All symlinks between the two are removed.

When a driver is removed, the list of devices that it supports is iterated over, and the driver' s remove callback is called for each one. The device is removed from that list and the symlinks removed.

1.2 Bus Types

1.2.1 Definition

See the kerneldoc for the struct bus_type.

int bus_register(struct bus_type * bus);

1.2.2 Declaration

Each bus type in the kernel (PCI, USB, etc) should declare one static object of this type. They must initialize the name field, and may optionally initialize the match callback:

```
struct bus_type pci_bus_type = {
       .name = "pci",
       .match        = pci_bus_match,
};
```

The structure should be exported to drivers in a header file:

extern struct bus_type pci_bus_type;

1.2.3 Registration

When a bus driver is initialized, it calls bus_register. This initializes the rest of the fields in the bus object and inserts it into a global list of bus types. Once the bus object is registered, the fields in it are usable by the bus driver.

1.2.4 Callbacks

1.2.5 match(): Attaching Drivers to Devices

The format of device ID structures and the semantics for comparing them are inherently bus-specific. Drivers typically declare an array of device IDs of devices they support that reside in a bus-specific driver structure.

The purpose of the match callback is to give the bus an opportunity to determine if a particular driver supports a particular device by comparing the device IDs the driver supports with the device ID of a particular device, without sacrificing bus-specific functionality or type-safety.

When a driver is registered with the bus, the bus' s list of devices is iterated over, and the match callback is called for each device that does not have a driver associated with it.

1.2.6 Device and Driver Lists

The lists of devices and drivers are intended to replace the local lists that many buses keep. They are lists of struct devices and struct device_drivers, respectively. Bus drivers are free to use the lists as they please, but conversion to the bus-specific type may be necessary.

The LDM core provides helper functions for iterating over each list:

```
int bus_for_each_dev(struct bus_type * bus, struct device * start,
                     void * data,
                     int (*fn)(struct device *, void *));

int bus_for_each_drv(struct bus_type * bus, struct device_driver * start,
                     void * data, int (*fn)(struct device_driver *, void␣
↪*));
```

These helpers iterate over the respective list, and call the callback for each device or driver in the list. All list accesses are synchronized by taking the bus's lock (read currently). The reference count on each object in the list is incremented before the callback is called; it is decremented after the next object has been obtained. The lock is not held when calling the callback.

1.2.7 sysfs

There is a top-level directory named 'bus'.

Each bus gets a directory in the bus directory, along with two default directories:

```
/sys/bus/pci/
|-- devices
`-- drivers
```

Drivers registered with the bus get a directory in the bus's drivers directory:

```
/sys/bus/pci/
|-- devices
`-- drivers
    |-- Intel ICH
    |-- Intel ICH Joystick
    |-- agpgart
    `-- e100
```

Each device that is discovered on a bus of that type gets a symlink in the bus's devices directory to the device's directory in the physical hierarchy:

```
/sys/bus/pci/
|-- devices
|   |-- 00:00.0 -> ../../../root/pci0/00:00.0
|   |-- 00:01.0 -> ../../../root/pci0/00:01.0
|   `-- 00:02.0 -> ../../../root/pci0/00:02.0
`-- drivers
```

1.2.8 Exporting Attributes

```
struct bus_attribute {
      struct attribute        attr;
      ssize_t (*show)(struct bus_type *, char * buf);
      ssize_t (*store)(struct bus_type *, const char * buf, size_t count);
};
```

Bus drivers can export attributes using the BUS_ATTR_RW macro that works similarly to the DEVICE_ATTR_RW macro for devices. For example, a definition like this:

```
static BUS_ATTR_RW(debug);
```

is equivalent to declaring:

```
static bus_attribute bus_attr_debug;
```

This can then be used to add and remove the attribute from the bus's sysfs directory using:

```
int bus_create_file(struct bus_type *, struct bus_attribute *);
void bus_remove_file(struct bus_type *, struct bus_attribute *);
```

1.3 Device Classes

1.3.1 Introduction

A device class describes a type of device, like an audio or network device. The following device classes have been identified:

<Insert List of Device Classes Here>

Each device class defines a set of semantics and a programming interface that devices of that class adhere to. Device drivers are the implementation of that programming interface for a particular device on a particular bus.

Device classes are agnostic with respect to what bus a device resides on.

1.3.2 Programming Interface

The device class structure looks like:

```
typedef int (*devclass_add)(struct device *);
typedef void (*devclass_remove)(struct device *);
```

See the kerneldoc for the struct class.

A typical device class definition would look like:

```
struct device_class input_devclass = {
      .name           = "input",
      .add_device     = input_add_device,
      .remove_device  = input_remove_device,
};
```

Each device class structure should be exported in a header file so it can be used by drivers, extensions and interfaces.

Device classes are registered and unregistered with the core using:

```
int devclass_register(struct device_class * cls);
void devclass_unregister(struct device_class * cls);
```

1.3.3 Devices

As devices are bound to drivers, they are added to the device class that the driver belongs to. Before the driver model core, this would typically happen during the driver's probe() callback, once the device has been initialized. It now happens after the probe() callback finishes from the core.

The device is enumerated in the class. Each time a device is added to the class, the class's devnum field is incremented and assigned to the device. The field is never decremented, so if the device is removed from the class and re-added, it will receive a different enumerated value.

The class is allowed to create a class-specific structure for the device and store it in the device's class_data pointer.

There is no list of devices in the device class. Each driver has a list of devices that it supports. The device class has a list of drivers of that particular class. To access all of the devices in the class, iterate over the device lists of each driver in the class.

1.3.4 Device Drivers

Device drivers are added to device classes when they are registered with the core. A driver specifies the class it belongs to by setting the struct device_driver::devclass field.

1.3.5 sysfs directory structure

There is a top-level sysfs directory named 'class'.

Each class gets a directory in the class directory, along with two default subdirectories:

```
class/
`-- input
    |-- devices
    `-- drivers
```

Drivers registered with the class get a symlink in the drivers/ directory that points to the driver's directory (under its bus directory):

```
class/
`-- input
    |-- devices
    `-- drivers
        `-- usb:usb_mouse -> ../../../bus/drivers/usb_mouse/
```

Each device gets a symlink in the devices/ directory that points to the device's directory in the physical hierarchy:

```
class/
`-- input
    |-- devices
    |   `-- 1 -> ../../../root/pci0/00:1f.0/usb_bus/00:1f.2-1:0/
    `-- drivers
```

1.3.6 Exporting Attributes

```
struct devclass_attribute {
      struct attribute        attr;
      ssize_t (*show)(struct device_class *, char * buf, size_t count,␣
↪loff_t off);
      ssize_t (*store)(struct device_class *, const char * buf, size_t␣
↪count, loff_t off);
};
```

Class drivers can export attributes using the DEVCLASS_ATTR macro that works similarly to the DEVICE_ATTR macro for devices. For example, a definition like this:

```
static DEVCLASS_ATTR(debug,0644,show_debug,store_debug);
```

is equivalent to declaring:

```
static devclass_attribute devclass_attr_debug;
```

The bus driver can add and remove the attribute from the class's sysfs directory using:

```
int devclass_create_file(struct device_class *, struct devclass_attribute␣
↪*);
void devclass_remove_file(struct device_class *, struct devclass_attribute␣
↪*);
```

In the example above, the file will be named 'debug' in placed in the class's directory in sysfs.

1.3.7 Interfaces

There may exist multiple mechanisms for accessing the same device of a particular class type. Device interfaces describe these mechanisms.

When a device is added to a device class, the core attempts to add it to every interface that is registered with the device class.

1.4 Device Driver Design Patterns

This document describes a few common design patterns found in device drivers. It is likely that subsystem maintainers will ask driver developers to conform to these design patterns.

1. State Container
2. `container_of()`

1.4.1 1. State Container

While the kernel contains a few device drivers that assume that they will only be probed() once on a certain system (singletons), it is custom to assume that the device the driver binds to will appear in several instances. This means that the probe() function and all callbacks need to be reentrant.

The most common way to achieve this is to use the state container design pattern. It usually has this form:

```
struct foo {
    spinlock_t lock; /* Example member */
    (...)
};

static int foo_probe(...)
{
    struct foo *foo;

    foo = devm_kzalloc(dev, sizeof(*foo), GFP_KERNEL);
    if (!foo)
        return -ENOMEM;
    spin_lock_init(&foo->lock);
    (...)
}
```

This will create an instance of struct foo in memory every time probe() is called. This is our state container for this instance of the device driver. Of course it is then necessary to always pass this instance of the state around to all functions that need access to the state and its members.

For example, if the driver is registering an interrupt handler, you would pass around a pointer to struct foo like this:

```
static irqreturn_t foo_handler(int irq, void *arg)
{
    struct foo *foo = arg;
    (...)
}

static int foo_probe(...)
{
    struct foo *foo;

    (...)
    ret = request_irq(irq, foo_handler, 0, "foo", foo);
}
```

This way you always get a pointer back to the correct instance of foo in your interrupt handler.

1.4.2 2. container_of()

Continuing on the above example we add an offloaded work:

```
struct foo {
    spinlock_t lock;
    struct workqueue_struct *wq;
    struct work_struct offload;
    (...)
};

static void foo_work(struct work_struct *work)
{
    struct foo *foo = container_of(work, struct foo, offload);

    (...)
}

static irqreturn_t foo_handler(int irq, void *arg)
{
    struct foo *foo = arg;

    queue_work(foo->wq, &foo->offload);
    (...)
}

static int foo_probe(...)
{
    struct foo *foo;

    foo->wq = create_singlethread_workqueue("foo-wq");
    INIT_WORK(&foo->offload, foo_work);
    (...)
}
```

The design pattern is the same for an hrtimer or something similar that will return a single argument which is a pointer to a struct member in the callback.

`container_of()` is a macro defined in <linux/kernel.h>

What container_of() does is to obtain a pointer to the containing struct from a pointer to a member by a simple subtraction using the offsetof() macro from standard C, which allows something similar to object oriented behaviours. Notice that the contained member must not be a pointer, but an actual member for this to work.

We can see here that we avoid having global pointers to our struct foo * instance this way, while still keeping the number of parameters passed to the work function to a single pointer.

1.5 The Basic Device Structure

See the kerneldoc for the struct device.

1.5.1 Programming Interface

The bus driver that discovers the device uses this to register the device with the core:

```
int device_register(struct device * dev);
```

The bus should initialize the following fields:

- parent
- name
- bus_id
- bus

A device is removed from the core when its reference count goes to 0. The reference count can be adjusted using:

```
struct device * get_device(struct device * dev);
void put_device(struct device * dev);
```

get_device() will return a pointer to the struct device passed to it if the reference is not already 0 (if it' s in the process of being removed already).

A driver can access the lock in the device structure using:

```
void lock_device(struct device * dev);
void unlock_device(struct device * dev);
```

1.5.2 Attributes

```
struct device_attribute {
      struct attribute        attr;
      ssize_t (*show)(struct device *dev, struct device_attribute *attr,
                      char *buf);
      ssize_t (*store)(struct device *dev, struct device_attribute *attr,
                       const char *buf, size_t count);
};
```

Attributes of devices can be exported by a device driver through sysfs.

Please see Documentation/filesystems/sysfs.rst for more information on how sysfs works.

As explained in Documentation/core-api/kobject.rst, device attributes must be created before the KOBJ_ADD uevent is generated. The only way to realize that is by defining an attribute group.

Attributes are declared using a macro called DEVICE_ATTR:

```
#define DEVICE_ATTR(name,mode,show,store)
```

Example::

```
static DEVICE_ATTR(type, 0444, show_type, NULL);
static DEVICE_ATTR(power, 0644, show_power, store_power);
```

This declares two structures of type struct device_attribute with respective names ‘dev_attr_type’ and ‘dev_attr_power’. These two attributes can be organized as follows into a group:

```
static struct attribute *dev_attrs[] = {
      &dev_attr_type.attr,
      &dev_attr_power.attr,
      NULL,
};

static struct attribute_group dev_attr_group = {
      .attrs = dev_attrs,
};

static const struct attribute_group *dev_attr_groups[] = {
      &dev_attr_group,
      NULL,
};
```

This array of groups can then be associated with a device by setting the group pointer in struct device before `device_register()` is invoked:

```
dev->groups = dev_attr_groups;
device_register(dev);
```

The `device_register()` function will use the ‘groups’ pointer to create the device attributes and the `device_unregister()` function will use this pointer to remove the device attributes.

Word of warning: While the kernel allows `device_create_file()` and `device_remove_file()` to be called on a device at any time, userspace has strict expectations on when attributes get created. When a new device is registered in the kernel, a uevent is generated to notify userspace (like udev) that a new device is available. If attributes are added after the device is registered, then userspace won't get notified and userspace will not know about the new attributes.

This is important for device driver that need to publish additional attributes for a device at driver probe time. If the device driver simply calls `device_create_file()` on the device structure passed to it, then userspace will never be notified of the new attributes.

1.6 Devres - Managed Device Resource

Tejun Heo <teheo@suse.de>

First draft 10 January 2007

1.6.1 1. Intro

devres came up while trying to convert libata to use iomap. Each iomapped address should be kept and unmapped on driver detach. For example, a plain SFF ATA controller (that is, good old PCI IDE) in native mode makes use of 5 PCI BARs and all of them should be maintained.

As with many other device drivers, libata low level drivers have sufficient bugs in ->remove and ->probe failure path. Well, yes, that's probably because libata low level driver developers are lazy bunch, but aren't all low level driver developers? After spending a day fiddling with braindamaged hardware with no document or braindamaged document, if it's finally working, well, it's working.

For one reason or another, low level drivers don't receive as much attention or testing as core code, and bugs on driver detach or initialization failure don't happen often enough to be noticeable. Init failure path is worse because it's much less travelled while needs to handle multiple entry points.

So, many low level drivers end up leaking resources on driver detach and having half broken failure path implementation in ->probe() which would leak resources or even cause oops when failure occurs. iomap adds more to this mix. So do msi and msix.

1.6.2 2. Devres

devres is basically linked list of arbitrarily sized memory areas associated with a struct device. Each devres entry is associated with a release function. A devres can be released in several ways. No matter what, all devres entries are released on driver detach. On release, the associated release function is invoked and then the devres entry is freed.

Managed interface is created for resources commonly used by device drivers using devres. For example, coherent DMA memory is acquired using

dma_alloc_coherent(). The managed version is called dmam_alloc_coherent(). It is identical to dma_alloc_coherent() except for the DMA memory allocated using it is managed and will be automatically released on driver detach. Implementation looks like the following:

```
struct dma_devres {
      size_t          size;
      void            *vaddr;
      dma_addr_t      dma_handle;
};

static void dmam_coherent_release(struct device *dev, void *res)
{
      struct dma_devres *this = res;

      dma_free_coherent(dev, this->size, this->vaddr, this->dma_handle);
}

dmam_alloc_coherent(dev, size, dma_handle, gfp)
{
      struct dma_devres *dr;
      void *vaddr;

      dr = devres_alloc(dmam_coherent_release, sizeof(*dr), gfp);
      ...

      /* alloc DMA memory as usual */
      vaddr = dma_alloc_coherent(...);
      ...

      /* record size, vaddr, dma_handle in dr */
      dr->vaddr = vaddr;
      ...

      devres_add(dev, dr);

      return vaddr;
}
```

If a driver uses dmam_alloc_coherent(), the area is guaranteed to be freed whether initialization fails half-way or the device gets detached. If most resources are acquired using managed interface, a driver can have much simpler init and exit code. Init path basically looks like the following:

```
my_init_one()
{
      struct mydev *d;

      d = devm_kzalloc(dev, sizeof(*d), GFP_KERNEL);
      if (!d)
              return -ENOMEM;

      d->ring = dmam_alloc_coherent(...);
      if (!d->ring)
              return -ENOMEM;
```

(continues on next page)

(continued from previous page)

```
      if (check something)
              return -EINVAL;
      ...

      return register_to_upper_layer(d);
}
```

And exit path:

```
my_remove_one()
{
      unregister_from_upper_layer(d);
      shutdown_my_hardware();
}
```

As shown above, low level drivers can be simplified a lot by using devres. Complexity is shifted from less maintained low level drivers to better maintained higher layer. Also, as init failure path is shared with exit path, both can get more testing.

Note though that when converting current calls or assignments to managed devm_* versions it is up to you to check if internal operations like allocating memory, have failed. Managed resources pertains to the freeing of these resources only - all other checks needed are still on you. In some cases this may mean introducing checks that were not necessary before moving to the managed devm_* calls.

1.6.3 3. Devres group

Devres entries can be grouped using devres group. When a group is released, all contained normal devres entries and properly nested groups are released. One usage is to rollback series of acquired resources on failure. For example:

```
 if (!devres_open_group(dev, NULL, GFP_KERNEL))
       return -ENOMEM;

 acquire A;
 if (failed)
       goto err;

 acquire B;
 if (failed)
       goto err;
 ...

 devres_remove_group(dev, NULL);
 return 0;

err:
 devres_release_group(dev, NULL);
 return err_code;
```

As resource acquisition failure usually means probe failure, constructs like above are usually useful in midlayer driver (e.g. libata core layer) where interface func-

tion shouldn't have side effect on failure. For LLDs, just returning error code suffices in most cases.

Each group is identified by void *id. It can either be explicitly specified by @id argument to `devres_open_group()` or automatically created by passing NULL as @id as in the above example. In both cases, `devres_open_group()` returns the group's id. The returned id can be passed to other devres functions to select the target group. If NULL is given to those functions, the latest open group is selected.

For example, you can do something like the following:

```
int my_midlayer_create_something()
{
      if (!devres_open_group(dev, my_midlayer_create_something, GFP_
↪KERNEL))
              return -ENOMEM;

      ...

      devres_close_group(dev, my_midlayer_create_something);
      return 0;
}

void my_midlayer_destroy_something()
{
      devres_release_group(dev, my_midlayer_create_something);
}
```

1.6.4 4. Details

Lifetime of a devres entry begins on devres allocation and finishes when it is released or destroyed (removed and freed) - no reference counting.

devres core guarantees atomicity to all basic devres operations and has support for single-instance devres types (atomic lookup-and-add-if-not-found). Other than that, synchronizing concurrent accesses to allocated devres data is caller's responsibility. This is usually non-issue because bus ops and resource allocations already do the job.

For an example of single-instance devres type, read pcim_iomap_table() in lib/devres.c.

All devres interface functions can be called without context if the right gfp mask is given.

1.6.5 5. Overhead

Each devres bookkeeping info is allocated together with requested data area. With debug option turned off, bookkeeping info occupies 16 bytes on 32bit machines and 24 bytes on 64bit (three pointers rounded up to ull alignment). If singly linked list is used, it can be reduced to two pointers (8 bytes on 32bit, 16 bytes on 64bit).

Each devres group occupies 8 pointers. It can be reduced to 6 if singly linked list is used.

Memory space overhead on ahci controller with two ports is between 300 and 400 bytes on 32bit machine after naive conversion (we can certainly invest a bit more effort into libata core layer).

1.6.6 6. List of managed interfaces

CLOCK devm_clk_get() devm_clk_get_optional() devm_clk_put() devm_clk_bulk_get() devm_clk_bulk_get_all() devm_clk_bulk_get_optional() devm_get_clk_from_childl() devm_clk_hw_register() devm_of_clk_add_hw_provider() devm_clk_hw_register_clkdev()

DMA dmaenginem_async_device_register() dmam_alloc_coherent() dmam_alloc_attrs() dmam_free_coherent() dmam_pool_create() dmam_pool_destroy()

DRM devm_drm_dev_init()

GPIO devm_gpiod_get() devm_gpiod_get_array() devm_gpiod_get_array_optional() devm_gpiod_get_index() devm_gpiod_get_index_optional() devm_gpiod_get_optional() devm_gpiod_put() devm_gpiod_unhinge() devm_gpiochip_add_data() devm_gpio_request() devm_gpio_request_one() devm_gpio_free()

I2C devm_i2c_new_dummy_device()

IIO devm_iio_device_alloc() devm_iio_device_register() devm_iio_kfifo_allocate() devm_iio_triggered_buffer_setup() devm_iio_trigger_alloc() devm_iio_trigger_register() devm_iio_channel_get() devm_iio_channel_get_all()

INPUT devm_input_allocate_device()

IO region devm_release_mem_region() devm_release_region() devm_release_resource() devm_request_mem_region() devm_request_region() devm_request_resource()

IOMAP devm_ioport_map() devm_ioport_unmap() devm_ioremap() devm_ioremap_uc() devm_ioremap_wc() devm_ioremap_resource() : checks resource, requests memory region, ioremaps devm_ioremap_resource_wc() devm_platform_ioremap_resource() : calls devm_ioremap_resource() for platform device devm_platform_ioremap_resource_wc() devm_platform_ioremap_resource_byname() devm_platform_get_and_ioremap_resour devm_iounmap() pcim_iomap() pcim_iomap_regions() : do request_region() and iomap() on multiple BARs pcim_iomap_table() : array of mapped addresses indexed by BAR pcim_iounmap()

IRQ devm_free_irq() devm_request_any_context_irq() devm_request_irq() devm_request_threaded_irq() devm_irq_alloc_descs() devm_irq_alloc_desc() devm_irq_alloc_desc_at() devm_irq_alloc_desc_from() devm_irq_alloc_descs_from() devm_irq_alloc_generic_chip() devm_irq_setup_generic_chip() devm_irq_sim_init()

LED devm_led_classdev_register() devm_led_classdev_unregister()

MDIO devm_mdiobus_alloc() devm_mdiobus_alloc_size() devm_mdiobus_free()

MEM `devm_free_pages()` `devm_get_free_pages()` `devm_kasprintf()` devm_kcalloc() `devm_kfree()` `devm_kmalloc()` devm_kmalloc_array() `devm_kmemdup()` `devm_kstrdup()` `devm_kvasprintf()` devm_kzalloc()

MFD devm_mfd_add_devices()

MUX devm_mux_chip_alloc() devm_mux_chip_register() devm_mux_control_get()

NET devm_alloc_etherdev() devm_alloc_etherdev_mqs() devm_register_netdev()

PER-CPU MEM `devm_alloc_percpu()` `devm_free_percpu()`

PCI devm_pci_alloc_host_bridge() : managed PCI host bridge allocation `devm_pci_remap_cfgspace()` : ioremap PCI configuration space `devm_pci_remap_cfg_resource()` : ioremap PCI configuration space resource `pcim_enable_device()` : after success, all PCI ops become managed `pcim_pin_device()` : keep PCI device enabled after release

PHY devm_usb_get_phy() devm_usb_put_phy()

PINCTRL devm_pinctrl_get() devm_pinctrl_put() devm_pinctrl_register() devm_pinctrl_unregister()

POWER devm_reboot_mode_register() devm_reboot_mode_unregister()

PWM `devm_pwm_get()` `devm_pwm_put()`

REGULATOR devm_regulator_bulk_get() devm_regulator_get() devm_regulator_put() devm_regulator_register()

RESET devm_reset_control_get() devm_reset_controller_register()

SERDEV devm_serdev_device_open()

SLAVE DMA ENGINE devm_acpi_dma_controller_register()

SPI devm_spi_register_master()

WATCHDOG devm_watchdog_register_device()

1.7 Device Drivers

See the kerneldoc for the struct device_driver.

1.7.1 Allocation

Device drivers are statically allocated structures. Though there may be multiple devices in a system that a driver supports, struct device_driver represents the driver as a whole (not a particular device instance).

1.7.2 Initialization

The driver must initialize at least the name and bus fields. It should also initialize the devclass field (when it arrives), so it may obtain the proper linkage internally. It should also initialize as many of the callbacks as possible, though each is optional.

1.7.3 Declaration

As stated above, struct device_driver objects are statically allocated. Below is an example declaration of the eepro100 driver. This declaration is hypothetical only; it relies on the driver being converted completely to the new model:

```
static struct device_driver eepro100_driver = {
       .name           = "eepro100",
       .bus            = &pci_bus_type,

       .probe          = eepro100_probe,
       .remove                 = eepro100_remove,
       .suspend                = eepro100_suspend,
       .resume                 = eepro100_resume,
};
```

Most drivers will not be able to be converted completely to the new model because the bus they belong to has a bus-specific structure with bus-specific fields that cannot be generalized.

The most common example of this are device ID structures. A driver typically defines an array of device IDs that it supports. The format of these structures and the semantics for comparing device IDs are completely bus-specific. Defining them as bus-specific entities would sacrifice type-safety, so we keep bus-specific structures around.

Bus-specific drivers should include a generic struct device_driver in the definition of the bus-specific driver. Like this:

```
struct pci_driver {
       const struct pci_device_id *id_table;
       struct device_driver     driver;
};
```

A definition that included bus-specific fields would look like (using the eepro100 driver again):

```
static struct pci_driver eepro100_driver = {
       .id_table       = eepro100_pci_tbl,
       .driver                 = {
              .name           = "eepro100",
```

(continues on next page)

(continued from previous page)

```
                .bus            = &pci_bus_type,
                .probe          = eepro100_probe,
                .remove         = eepro100_remove,
                .suspend        = eepro100_suspend,
                .resume         = eepro100_resume,
        },
};
```

Some may find the syntax of embedded struct initialization awkward or even a bit ugly. So far, it's the best way we've found to do what we want…

1.7.4 Registration

```
int driver_register(struct device_driver *drv);
```

The driver registers the structure on startup. For drivers that have no bus-specific fields (i.e. don't have a bus-specific driver structure), they would use driver_register and pass a pointer to their struct device_driver object.

Most drivers, however, will have a bus-specific structure and will need to register with the bus using something like pci_driver_register.

It is important that drivers register their driver structure as early as possible. Registration with the core initializes several fields in the struct device_driver object, including the reference count and the lock. These fields are assumed to be valid at all times and may be used by the device model core or the bus driver.

1.7.5 Transition Bus Drivers

By defining wrapper functions, the transition to the new model can be made easier. Drivers can ignore the generic structure altogether and let the bus wrapper fill in the fields. For the callbacks, the bus can define generic callbacks that forward the call to the bus-specific callbacks of the drivers.

This solution is intended to be only temporary. In order to get class information in the driver, the drivers must be modified anyway. Since converting drivers to the new model should reduce some infrastructural complexity and code size, it is recommended that they are converted as class information is added.

1.7.6 Access

Once the object has been registered, it may access the common fields of the object, like the lock and the list of devices:

```
int driver_for_each_dev(struct device_driver *drv, void *data,
                        int (*callback)(struct device *dev, void *data));
```

The devices field is a list of all the devices that have been bound to the driver. The LDM core provides a helper function to operate on all the devices a driver controls. This helper locks the driver on each node access, and does proper reference counting on each device as it accesses it.

1.7.7 sysfs

When a driver is registered, a sysfs directory is created in its bus' s directory. In this directory, the driver can export an interface to userspace to control operation of the driver on a global basis; e.g. toggling debugging output in the driver.

A future feature of this directory will be a 'devices' directory. This directory will contain symlinks to the directories of devices it supports.

1.7.8 Callbacks

```
int     (*probe)        (struct device *dev);
```

The probe() entry is called in task context, with the bus' s rwsem locked and the driver partially bound to the device. Drivers commonly use `container_of()` to convert "dev" to a bus-specific type, both in probe() and other routines. That type often provides device resource data, such as pci_dev.resource[] or platform_device.resources, which is used in addition to dev->platform_data to initialize the driver.

This callback holds the driver-specific logic to bind the driver to a given device. That includes verifying that the device is present, that it' s a version the driver can handle, that driver data structures can be allocated and initialized, and that any hardware can be initialized. Drivers often store a pointer to their state with dev_set_drvdata(). When the driver has successfully bound itself to that device, then probe() returns zero and the driver model code will finish its part of binding the driver to that device.

A driver' s probe() may return a negative errno value to indicate that the driver did not bind to this device, in which case it should have released all resources it allocated.

Optionally, probe() may return -EPROBE_DEFER if the driver depends on resources that are not yet available (e.g., supplied by a driver that hasn' t initialized yet). The driver core will put the device onto the deferred probe list and will try to call it again later. If a driver must defer, it should return -EPROBE_DEFER as early as possible to reduce the amount of time spent on setup work that will need to be unwound and reexecuted at a later time.

> **Warning:** -EPROBE_DEFER must not be returned if probe() has already created child devices, even if those child devices are removed again in a cleanup path. If -EPROBE_DEFER is returned after a child device has been registered, it may result in an infinite loop of .probe() calls to the same driver.

```
void    (*sync_state)   (struct device *dev);
```

sync_state is called only once for a device. It' s called when all the consumer devices of the device have successfully probed. The list of consumers of the device is obtained by looking at the device links connecting that device to its consumer devices.

The first attempt to call sync_state() is made during late_initcall_sync() to give firmware and drivers time to link devices to each other. During the first attempt at calling sync_state(), if all the consumers of the device at that point in time have already probed successfully, sync_state() is called right away. If there are no consumers of the device during the first attempt, that too is considered as "all consumers of the device have probed" and sync_state() is called right away.

If during the first attempt at calling sync_state() for a device, there are still consumers that haven't probed successfully, the sync_state() call is postponed and reattempted in the future only when one or more consumers of the device probe successfully. If during the reattempt, the driver core finds that there are one or more consumers of the device that haven't probed yet, then sync_state() call is postponed again.

A typical use case for sync_state() is to have the kernel cleanly take over management of devices from the bootloader. For example, if a device is left on and at a particular hardware configuration by the bootloader, the device's driver might need to keep the device in the boot configuration until all the consumers of the device have probed. Once all the consumers of the device have probed, the device's driver can synchronize the hardware state of the device to match the aggregated software state requested by all the consumers. Hence the name sync_state().

While obvious examples of resources that can benefit from sync_state() include resources such as regulator, sync_state() can also be useful for complex resources like IOMMUs. For example, IOMMUs with multiple consumers (devices whose addresses are remapped by the IOMMU) might need to keep their mappings fixed at (or additive to) the boot configuration until all its consumers have probed.

While the typical use case for sync_state() is to have the kernel cleanly take over management of devices from the bootloader, the usage of sync_state() is not restricted to that. Use it whenever it makes sense to take an action after all the consumers of a device have probed:

```
::
```

int (*remove) (struct device *dev);

remove is called to unbind a driver from a device. This may be called if a device is physically removed from the system, if the driver module is being unloaded, during a reboot sequence, or in other cases.

It is up to the driver to determine if the device is present or not. It should free any resources allocated specifically for the device; i.e. anything in the device's driver_data field.

If the device is still present, it should quiesce the device and place it into a supported low-power state.

```
int     (*suspend)      (struct device *dev, pm_message_t state);
```

suspend is called to put the device in a low power state.

```
int     (*resume)       (struct device *dev);
```

Resume is used to bring a device back from a low power state.

1.7.9 Attributes

```
struct driver_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device_driver *driver, char *buf);
        ssize_t (*store)(struct device_driver *, const char *buf, size_t␣
↪count);
};
```

Device drivers can export attributes via their sysfs directories. Drivers can declare attributes using a DRIVER_ATTR_RW and DRIVER_ATTR_RO macro that works identically to the DEVICE_ATTR_RW and DEVICE_ATTR_RO macros.

Example:

```
DRIVER_ATTR_RW(debug);
```

This is equivalent to declaring:

```
struct driver_attribute driver_attr_debug;
```

This can then be used to add and remove the attribute from the driver's directory using:

```
int driver_create_file(struct device_driver *, const struct driver_
↪attribute *);
void driver_remove_file(struct device_driver *, const struct driver_
↪attribute *);
```

1.8 The Linux Kernel Device Model

Patrick Mochel <mochel@digitalimplant.org>

Drafted 26 August 2002 Updated 31 January 2006

1.8.1 Overview

The Linux Kernel Driver Model is a unification of all the disparate driver models that were previously used in the kernel. It is intended to augment the bus-specific drivers for bridges and devices by consolidating a set of data and operations into globally accessible data structures.

Traditional driver models implemented some sort of tree-like structure (sometimes just a list) for the devices they control. There wasn't any uniformity across the different bus types.

The current driver model provides a common, uniform data model for describing a bus and the devices that can appear under the bus. The unified bus model includes a set of common attributes which all busses carry, and a set of common callbacks, such as device discovery during bus probing, bus shutdown, bus power management, etc.

The common device and bridge interface reflects the goals of the modern computer: namely the ability to do seamless device “plug and play”, power management, and hot plug. In particular, the model dictated by Intel and Microsoft (namely ACPI) ensures that almost every device on almost any bus on an x86-compatible system can work within this paradigm. Of course, not every bus is able to support all such operations, although most buses support most of those operations.

1.8.2 Downstream Access

Common data fields have been moved out of individual bus layers into a common data structure. These fields must still be accessed by the bus layers, and sometimes by the device-specific drivers.

Other bus layers are encouraged to do what has been done for the PCI layer. struct pci_dev now looks like this:

```
struct pci_dev {
      ...

      struct device dev;     /* Generic device interface */
      ...
};
```

Note first that the struct device dev within the struct pci_dev is statically allocated. This means only one allocation on device discovery.

Note also that that struct device dev is not necessarily defined at the front of the pci_dev structure. This is to make people think about what they’re doing when switching between the bus driver and the global driver, and to discourage meaningless and incorrect casts between the two.

The PCI bus layer freely accesses the fields of struct device. It knows about the structure of struct pci_dev, and it should know the structure of struct device. Individual PCI device drivers that have been converted to the current driver model generally do not and should not touch the fields of struct device, unless there is a compelling reason to do so.

The above abstraction prevents unnecessary pain during transitional phases. If it were not done this way, then when a field was renamed or removed, every downstream driver would break. On the other hand, if only the bus layer (and not the device layer) accesses the struct device, it is only the bus layer that needs to change.

1.8.3 User Interface

By virtue of having a complete hierarchical view of all the devices in the system, exporting a complete hierarchical view to userspace becomes relatively easy. This has been accomplished by implementing a special purpose virtual file system named sysfs.

Almost all mainstream Linux distros mount this filesystem automatically; you can see some variation of the following in the output of the "mount" command:

```
$ mount
...
none on /sys type sysfs (rw,noexec,nosuid,nodev)
...
$
```

The auto-mounting of sysfs is typically accomplished by an entry similar to the following in the /etc/fstab file:

```
none          /sys    sysfs    defaults               0 0
```

or something similar in the /lib/init/fstab file on Debian-based systems:

```
none            /sys    sysfs    nodev,noexec,nosuid    0 0
```

If sysfs is not automatically mounted, you can always do it manually with:

```
# mount -t sysfs sysfs /sys
```

Whenever a device is inserted into the tree, a directory is created for it. This directory may be populated at each layer of discovery - the global layer, the bus layer, or the device layer.

The global layer currently creates two files - 'name' and 'power'. The former only reports the name of the device. The latter reports the current power state of the device. It will also be used to set the current power state.

The bus layer may also create files for the devices it finds while probing the bus. For example, the PCI layer currently creates 'irq' and 'resource' files for each PCI device.

A device-specific driver may also export files in its directory to expose device-specific data or tunable interfaces.

More information about the sysfs directory layout can be found in the other documents in this directory and in the file Documentation/filesystems/sysfs.rst.

1.9 Platform Devices and Drivers

See <linux/platform_device.h> for the driver model interface to the platform bus: platform_device, and platform_driver. This pseudo-bus is used to connect devices on busses with minimal infrastructure, like those used to integrate peripherals on many system-on-chip processors, or some "legacy" PC interconnects; as opposed to large formally specified ones like PCI or USB.

1.9.1 Platform devices

Platform devices are devices that typically appear as autonomous entities in the system. This includes legacy port-based devices and host bridges to peripheral buses, and most controllers integrated into system-on-chip platforms. What they usually have in common is direct addressing from a CPU bus. Rarely, a platform_device will be connected through a segment of some other kind of bus; but its registers will still be directly addressable.

Platform devices are given a name, used in driver binding, and a list of resources such as addresses and IRQs:

```
struct platform_device {
      const char      *name;
      u32             id;
      struct device   dev;
      u32             num_resources;
      struct resource *resource;
};
```

1.9.2 Platform drivers

Platform drivers follow the standard driver model convention, where discovery/enumeration is handled outside the drivers, and drivers provide probe() and remove() methods. They support power management and shutdown notifications using the standard conventions:

```
struct platform_driver {
      int (*probe)(struct platform_device *);
      int (*remove)(struct platform_device *);
      void (*shutdown)(struct platform_device *);
      int (*suspend)(struct platform_device *, pm_message_t state);
      int (*suspend_late)(struct platform_device *, pm_message_t state);
      int (*resume_early)(struct platform_device *);
      int (*resume)(struct platform_device *);
      struct device_driver driver;
};
```

Note that probe() should in general verify that the specified device hardware actually exists; sometimes platform setup code can't be sure. The probing can use device resources, including clocks, and device platform_data.

Platform drivers register themselves the normal way:

```
int platform_driver_register(struct platform_driver *drv);
```

Or, in common situations where the device is known not to be hot-pluggable, the probe() routine can live in an init section to reduce the driver's runtime memory footprint:

```
int platform_driver_probe(struct platform_driver *drv,
                  int (*probe)(struct platform_device *))
```

Kernel modules can be composed of several platform drivers. The platform core provides helpers to register and unregister an array of drivers:

```
int __platform_register_drivers(struct platform_driver * const *drivers,
                                unsigned int count, struct module *owner);
void platform_unregister_drivers(struct platform_driver * const *drivers,
                                 unsigned int count);
```

If one of the drivers fails to register, all drivers registered up to that point will be unregistered in reverse order. Note that there is a convenience macro that passes THIS_MODULE as owner parameter:

```
#define platform_register_drivers(drivers, count)
```

1.9.3 Device Enumeration

As a rule, platform specific (and often board-specific) setup code will register platform devices:

```
int platform_device_register(struct platform_device *pdev);

int platform_add_devices(struct platform_device **pdevs, int ndev);
```

The general rule is to register only those devices that actually exist, but in some cases extra devices might be registered. For example, a kernel might be configured to work with an external network adapter that might not be populated on all boards, or likewise to work with an integrated controller that some boards might not hook up to any peripherals.

In some cases, boot firmware will export tables describing the devices that are populated on a given board. Without such tables, often the only way for system setup code to set up the correct devices is to build a kernel for a specific target board. Such board-specific kernels are common with embedded and custom systems development.

In many cases, the memory and IRQ resources associated with the platform device are not enough to let the device's driver work. Board setup code will often provide additional information using the device's platform_data field to hold additional information.

Embedded systems frequently need one or more clocks for platform devices, which are normally kept off until they're actively needed (to save power). System setup also associates those clocks with the device, so that that calls to clk_get(&pdev->dev, clock_name) return them as needed.

1.9.4 Legacy Drivers: Device Probing

Some drivers are not fully converted to the driver model, because they take on a non-driver role: the driver registers its platform device, rather than leaving that for system infrastructure. Such drivers can't be hotplugged or coldplugged, since those mechanisms require device creation to be in a different system component than the driver.

The only "good" reason for this is to handle older system designs which, like original IBM PCs, rely on error-prone "probe-the-hardware" models for hardware configuration. Newer systems have largely abandoned that model, in favor of bus-level support for dynamic configuration (PCI, USB), or device tables provided by the boot firmware (e.g. PNPACPI on x86). There are too many conflicting options about what might be where, and even educated guesses by an operating system will be wrong often enough to make trouble.

This style of driver is discouraged. If you're updating such a driver, please try to move the device enumeration to a more appropriate location, outside the driver. This will usually be cleanup, since such drivers tend to already have "normal" modes, such as ones using device nodes that were created by PNP or by platform device setup.

None the less, there are some APIs to support such legacy drivers. Avoid using these calls except with such hotplug-deficient drivers:

```
struct platform_device *platform_device_alloc(
                const char *name, int id);
```

You can use `platform_device_alloc()` to dynamically allocate a device, which you will then initialize with resources and `platform_device_register()`. A better solution is usually:

```
struct platform_device *platform_device_register_simple(
                const char *name, int id,
                struct resource *res, unsigned int nres);
```

You can use `platform_device_register_simple()` as a one-step call to allocate and register a device.

1.9.5 Device Naming and Driver Binding

The platform_device.dev.bus_id is the canonical name for the devices. It's built from two components:

- platform_device.name ⋯which is also used to for driver matching.
- platform_device.id ⋯the device instance number, or else "-1"to indicate there's only one.

These are concatenated, so name/id "serial"/0 indicates bus_id "serial.0", and "serial/3" indicates bus_id "serial.3"; both would use the platform_driver named "serial". While "my_rtc"/-1 would be bus_id "my_rtc" (no instance id) and use the platform_driver called "my_rtc".

Driver binding is performed automatically by the driver core, invoking driver probe() after finding a match between device and driver. If the probe() succeeds, the driver and device are bound as usual. There are three different ways to find such a match:

- Whenever a device is registered, the drivers for that bus are checked for matches. Platform devices should be registered very early during system boot.
- When a driver is registered using platform_driver_register(), all unbound devices on that bus are checked for matches. Drivers usually register later during booting, or by module loading.
- Registering a driver using platform_driver_probe() works just like using platform_driver_register(), except that the driver won't be probed later if another device registers. (Which is OK, since this interface is only for use with non-hotpluggable devices.)

1.9.6 Early Platform Devices and Drivers

The early platform interfaces provide platform data to platform device drivers early on during the system boot. The code is built on top of the early_param() command line parsing and can be executed very early on.

Example: "earlyprintk" class early serial console in 6 steps

1.9.7 1. Registering early platform device data

The architecture code registers platform device data using the function early_platform_add_devices(). In the case of early serial console this should be hardware configuration for the serial port. Devices registered at this point will later on be matched against early platform drivers.

1.9.8 2. Parsing kernel command line

The architecture code calls parse_early_param() to parse the kernel command line. This will execute all matching early_param() callbacks. User specified early platform devices will be registered at this point. For the early serial console case the user can specify port on the kernel command line as "earlyprintk=serial.0" where "earlyprintk" is the class string, "serial" is the name of the platform driver and 0 is the platform device id. If the id is -1 then the dot and the id can be omitted.

1.9.9 3. Installing early platform drivers belonging to a certain class

The architecture code may optionally force registration of all early platform drivers belonging to a certain class using the function early_platform_driver_register_all(). User specified devices from step 2 have priority over these. This step is omitted by the serial driver example since the early serial driver code should be disabled unless the user has specified port on the kernel command line.

1.9.10 4. Early platform driver registration

Compiled-in platform drivers making use of early_platform_init() are automatically registered during step 2 or 3. The serial driver example should use early_platform_init(“earlyprintk” , &platform_driver).

1.9.11 5. Probing of early platform drivers belonging to a certain class

The architecture code calls early_platform_driver_probe() to match registered early platform devices associated with a certain class with registered early platform drivers. Matched devices will get probed(). This step can be executed at any point during the early boot. As soon as possible may be good for the serial port case.

1.9.12 6. Inside the early platform driver probe()

The driver code needs to take special care during early boot, especially when it comes to memory allocation and interrupt registration. The code in the probe() function can use is_early_platform_device() to check if it is called at early platform device or at the regular platform device time. The early serial driver performs register_console() at this point.

For further information, see <linux/platform_device.h>.

1.10 Porting Drivers to the New Driver Model

Patrick Mochel

7 January 2003

Overview

Please refer to Documentation/driver-api/driver-model/*.rst for definitions of various driver types and concepts.

Most of the work of porting devices drivers to the new model happens at the bus driver layer. This was intentional, to minimize the negative effect on kernel drivers, and to allow a gradual transition of bus drivers.

In a nutshell, the driver model consists of a set of objects that can be embedded in larger, bus-specific objects. Fields in these generic objects can replace fields in the bus-specific objects.

The generic objects must be registered with the driver model core. By doing so, they will exported via the sysfs filesystem. sysfs can be mounted by doing:

```
# mount -t sysfs sysfs /sys
```

The Process

Step 0: Read include/linux/device.h for object and function definitions.

Step 1: Registering the bus driver.

- Define a struct bus_type for the bus driver:

```
struct bus_type pci_bus_type = {
      .name           = "pci",
};
```

- Register the bus type.

  This should be done in the initialization function for the bus type, which is usually the `module_init()`, or equivalent, function:

```
static int __init pci_driver_init(void)
{
        return bus_register(&pci_bus_type);
}

subsys_initcall(pci_driver_init);
```

  The bus type may be unregistered (if the bus driver may be compiled as a module) by doing:

```
bus_unregister(&pci_bus_type);
```

- Export the bus type for others to use.

  Other code may wish to reference the bus type, so declare it in a shared header file and export the symbol.

From include/linux/pci.h:

```
extern struct bus_type pci_bus_type;
```

From file the above code appears in:

```
EXPORT_SYMBOL(pci_bus_type);
```

- This will cause the bus to show up in /sys/bus/pci/ with two subdirectories: ‘devices’ and ‘drivers’ :

```
# tree -d /sys/bus/pci/
/sys/bus/pci/
|-- devices
`-- drivers
```

Step 2: Registering Devices.

struct device represents a single device. It mainly contains metadata describing the relationship the device has to other entities.

- Embed a struct device in the bus-specific device type:

```
struct pci_dev {
       ...
       struct  device  dev;            /* Generic device interface */
       ...
};
```

  It is recommended that the generic device not be the first item in the struct to discourage programmers from doing mindless casts between the object types. Instead macros, or inline functions, should be created to convert from the generic object type:

```
#define to_pci_dev(n) container_of(n, struct pci_dev, dev)

or

static inline struct pci_dev * to_pci_dev(struct kobject * kobj)
{
    return container_of(n, struct pci_dev, dev);
}
```

  This allows the compiler to verify type-safety of the operations that are performed (which is Good).

- Initialize the device on registration.

  When devices are discovered or registered with the bus type, the bus driver should initialize the generic device. The most important things to initialize are the bus_id, parent, and bus fields.

  The bus_id is an ASCII string that contains the device's address on the bus. The format of this string is bus-specific. This is necessary for representing devices in sysfs.

  parent is the physical parent of the device. It is important that the bus driver sets this field correctly.

  The driver model maintains an ordered list of devices that it uses for power management. This list must be in order to guarantee that devices are shutdown before their physical parents, and vice versa. The order of this list is determined by the parent of registered devices.

  Also, the location of the device's sysfs directory depends on a device's parent. sysfs exports a directory structure that mirrors the device hierarchy. Accurately setting the parent guarantees that sysfs will accurately represent the hierarchy.

  The device's bus field is a pointer to the bus type the device belongs to. This should be set to the bus_type that was declared and initialized before.

  Optionally, the bus driver may set the device's name and release fields.

  The name field is an ASCII string describing the device, like

"ATI Technologies Inc Radeon QD"

The release field is a callback that the driver model core calls when the device has been removed, and all references to it have been released. More on this in a moment.

- Register the device.

  Once the generic device has been initialized, it can be registered with the driver model core by doing:

```
device_register(&dev->dev);
```

  It can later be unregistered by doing:

```
device_unregister(&dev->dev);
```

  This should happen on buses that support hotpluggable devices. If a bus driver unregisters a device, it should not immediately free it. It should instead wait for the driver model core to call the device's release method, then free the bus-specific object. (There may be other code that is currently referencing the device structure, and it would be rude to free the device while that is happening).

  When the device is registered, a directory in sysfs is created. The PCI tree in sysfs looks like:

```
/sys/devices/pci0/
|-- 00:00.0
|-- 00:01.0
|   `-- 01:00.0
|-- 00:02.0
|   `-- 02:1f.0
|       `-- 03:00.0
|-- 00:1e.0
|   `-- 04:04.0
|-- 00:1f.0
|-- 00:1f.1
|   |-- ide0
|   |   |-- 0.0
|   |   `-- 0.1
|   `-- ide1
|       `-- 1.0
|-- 00:1f.2
|-- 00:1f.3
`-- 00:1f.5
```

  Also, symlinks are created in the bus's 'devices' directory that point to the device's directory in the physical hierarchy:

```
/sys/bus/pci/devices/
|-- 00:00.0 -> ../../../devices/pci0/00:00.0
|-- 00:01.0 -> ../../../devices/pci0/00:01.0
|-- 00:02.0 -> ../../../devices/pci0/00:02.0
|-- 00:1e.0 -> ../../../devices/pci0/00:1e.0
|-- 00:1f.0 -> ../../../devices/pci0/00:1f.0
|-- 00:1f.1 -> ../../../devices/pci0/00:1f.1
```

(continues on next page)

(continued from previous page)

```
|-- 00:1f.2 -> ../../../devices/pci0/00:1f.2
|-- 00:1f.3 -> ../../../devices/pci0/00:1f.3
|-- 00:1f.5 -> ../../../devices/pci0/00:1f.5
|-- 01:00.0 -> ../../../devices/pci0/00:01.0/01:00.0
|-- 02:1f.0 -> ../../../devices/pci0/00:02.0/02:1f.0
|-- 03:00.0 -> ../../../devices/pci0/00:02.0/02:1f.0/03:00.0
`-- 04:04.0 -> ../../../devices/pci0/00:1e.0/04:04.0
```

Step 3: Registering Drivers.

struct device_driver is a simple driver structure that contains a set of operations that the driver model core may call.

- Embed a struct device_driver in the bus-specific driver.

  Just like with devices, do something like:

```
struct pci_driver {
       ...
       struct device_driver    driver;
};
```

- Initialize the generic driver structure.

  When the driver registers with the bus (e.g. doing pci_register_driver()), initialize the necessary fields of the driver: the name and bus fields.

- Register the driver.

  After the generic driver has been initialized, call:

```
driver_register(&drv->driver);
```

  to register the driver with the core.

  When the driver is unregistered from the bus, unregister it from the core by doing:

```
driver_unregister(&drv->driver);
```

  Note that this will block until all references to the driver have gone away. Normally, there will not be any.

- Sysfs representation.

  Drivers are exported via sysfs in their bus' s 'driver' s directory. For example:

```
/sys/bus/pci/drivers/
|-- 3c59x
|-- Ensoniq AudioPCI
|-- agpgart-amdk7
|-- e100
`-- serial
```

Step 4: Define Generic Methods for Drivers.

struct device_driver defines a set of operations that the driver model core calls. Most of these operations are probably similar to operations the bus already defines

for drivers, but taking different parameters.

It would be difficult and tedious to force every driver on a bus to simultaneously convert their drivers to generic format. Instead, the bus driver should define single instances of the generic methods that forward call to the bus-specific drivers. For instance:

```
static int pci_device_remove(struct device * dev)
{
        struct pci_dev * pci_dev = to_pci_dev(dev);
        struct pci_driver * drv = pci_dev->driver;

        if (drv) {
                if (drv->remove)
                        drv->remove(pci_dev);
                pci_dev->driver = NULL;
        }
        return 0;
}
```

The generic driver should be initialized with these methods before it is registered:

```
/* initialize common driver fields */
drv->driver.name = drv->name;
drv->driver.bus = &pci_bus_type;
drv->driver.probe = pci_device_probe;
drv->driver.resume = pci_device_resume;
drv->driver.suspend = pci_device_suspend;
drv->driver.remove = pci_device_remove;

/* register with core */
driver_register(&drv->driver);
```

Ideally, the bus should only initialize the fields if they are not already set. This allows the drivers to implement their own generic methods.

Step 5: Support generic driver binding.

The model assumes that a device or driver can be dynamically registered with the bus at any time. When registration happens, devices must be bound to a driver, or drivers must be bound to all devices that it supports.

A driver typically contains a list of device IDs that it supports. The bus driver compares these IDs to the IDs of devices registered with it. The format of the device IDs, and the semantics for comparing them are bus-specific, so the generic model does attempt to generalize them.

Instead, a bus may supply a method in struct bus_type that does the comparison:

```
int (*match)(struct device * dev, struct device_driver * drv);
```

match should return positive value if the driver supports the device, and zero otherwise. It may also return error code (for example -EPROBE_DEFER) if determining that given driver supports the device is not possible.

When a device is registered, the bus' s list of drivers is iterated over. bus->match() is called for each one until a match is found.

When a driver is registered, the bus's list of devices is iterated over. bus->match() is called for each device that is not already claimed by a driver.

When a device is successfully bound to a driver, device->driver is set, the device is added to a per-driver list of devices, and a symlink is created in the driver's sysfs directory that points to the device's physical directory:

```
/sys/bus/pci/drivers/
|-- 3c59x
|   `-- 00:0b.0 -> ../../../../devices/pci0/00:0b.0
|-- Ensoniq AudioPCI
|-- agpgart-amdk7
|   `-- 00:00.0 -> ../../../../devices/pci0/00:00.0
|-- e100
|   `-- 00:0c.0 -> ../../../../devices/pci0/00:0c.0
`-- serial
```

This driver binding should replace the existing driver binding mechanism the bus currently uses.

Step 6: Supply a hotplug callback.

Whenever a device is registered with the driver model core, the userspace program /sbin/hotplug is called to notify userspace. Users can define actions to perform when a device is inserted or removed.

The driver model core passes several arguments to userspace via environment variables, including

- ACTION: set to 'add' or 'remove'
- DEVPATH: set to the device's physical path in sysfs.

A bus driver may also supply additional parameters for userspace to consume. To do this, a bus must implement the 'hotplug' method in struct bus_type:

```
int (*hotplug) (struct device *dev, char **envp,
                int num_envp, char *buffer, int buffer_size);
```

This is called immediately before /sbin/hotplug is executed.

Step 7: Cleaning up the bus driver.

The generic bus, device, and driver structures provide several fields that can replace those defined privately to the bus driver.

- Device list.

struct bus_type contains a list of all devices registered with the bus type. This includes all devices on all instances of that bus type. An internal list that the bus uses may be removed, in favor of using this one.

The core provides an iterator to access these devices:

```
int bus_for_each_dev(struct bus_type * bus, struct device * start,
                     void * data, int (*fn)(struct device *, void *));
```

- Driver list.

struct bus_type also contains a list of all drivers registered with it. An internal list of drivers that the bus driver maintains may be removed in favor of using the generic one.

The drivers may be iterated over, like devices:

```
int bus_for_each_drv(struct bus_type * bus, struct device_driver * start,
                     void * data, int (*fn)(struct device_driver *, void␣
↪*));
```

Please see drivers/base/bus.c for more information.

- rwsem

struct bus_type contains an rwsem that protects all core accesses to the device and driver lists. This can be used by the bus driver internally, and should be used when accessing the device or driver lists the bus maintains.

- Device and driver fields.

Some of the fields in struct device and struct device_driver duplicate fields in the bus-specific representations of these objects. Feel free to remove the bus-specific ones and favor the generic ones. Note though, that this will likely mean fixing up all the drivers that reference the bus-specific fields (though those should all be 1-line changes).

CHAPTER

TWO

DRIVER BASICS

2.1 Driver Entry and Exit points

module_init(x)
driver initialization entry point

Parameters

x function to be run at kernel boot time or module insertion

Description

`module_init()` will either be called during do_initcalls() (if builtin) or at module insertion time (if a module). There can only be one per module.

module_exit(x)
driver exit entry point

Parameters

x function to be run when driver is removed

Description

`module_exit()` will wrap the driver clean-up code with cleanup_module() when used with rmmod when the driver is a module. If the driver is statically compiled into the kernel, `module_exit()` has no effect. There can only be one per module.

2.2 Driver device table

struct **pci_device_id**
PCI device ID structure

Definition

```
struct pci_device_id {
  __u32 vendor, device;
  __u32 subvendor, subdevice;
  __u32 class, class_mask;
  kernel_ulong_t driver_data;
};
```

Members

vendor Vendor ID to match (or PCI_ANY_ID)

device Device ID to match (or PCI_ANY_ID)

subvendor Subsystem vendor ID to match (or PCI_ANY_ID)

subdevice Subsystem device ID to match (or PCI_ANY_ID)

class Device class, subclass, and "interface" to match. See Appendix D of the PCI Local Bus Spec or include/linux/pci_ids.h for a full list of classes. Most drivers do not need to specify class/class_mask as vendor/device is normally sufficient.

class_mask Limit which sub-fields of the class field are compared. See drivers/scsi/sym53c8xx_2/ for example of usage.

driver_data Data private to the driver. Most drivers don't need to use driver_data field. Best practice is to use driver_data as an index into a static list of equivalent device types, instead of using it as a pointer.

struct **usb_device_id**
identifies USB devices for probing and hotplugging

Definition

```
struct usb_device_id {
  __u16 match_flags;
  __u16 idVendor;
  __u16 idProduct;
  __u16 bcdDevice_lo;
  __u16 bcdDevice_hi;
  __u8 bDeviceClass;
  __u8 bDeviceSubClass;
  __u8 bDeviceProtocol;
  __u8 bInterfaceClass;
  __u8 bInterfaceSubClass;
  __u8 bInterfaceProtocol;
  __u8 bInterfaceNumber;
  kernel_ulong_t driver_info ;
};
```

Members

match_flags Bit mask controlling which of the other fields are used to match against new devices. Any field except for driver_info may be used, although some only make sense in conjunction with other fields. This is usually set by a USB_DEVICE_*() macro, which sets all other fields in this structure except for driver_info.

idVendor USB vendor ID for a device; numbers are assigned by the USB forum to its members.

idProduct Vendor-assigned product ID.

bcdDevice_lo Low end of range of vendor-assigned product version numbers. This is also used to identify individual product versions, for a range consisting of a single device.

bcdDevice_hi High end of version number range. The range of product versions is inclusive.

bDeviceClass Class of device; numbers are assigned by the USB forum. Products may choose to implement classes, or be vendor-specific. Device classes specify behavior of all the interfaces on a device.

bDeviceSubClass Subclass of device; associated with bDeviceClass.

bDeviceProtocol Protocol of device; associated with bDeviceClass.

bInterfaceClass Class of interface; numbers are assigned by the USB forum. Products may choose to implement classes, or be vendor-specific. Interface classes specify behavior only of a given interface; other interfaces may support other classes.

bInterfaceSubClass Subclass of interface; associated with bInterfaceClass.

bInterfaceProtocol Protocol of interface; associated with bInterfaceClass.

bInterfaceNumber Number of interface; composite devices may use fixed interface numbers to differentiate between vendor-specific interfaces.

driver_info Holds information used by the driver. Usually it holds a pointer to a descriptor understood by the driver, or perhaps device flags.

Description

In most cases, drivers will create a table of device IDs by using `USB_DEVICE()`, or similar macros designed for that purpose. They will then export it to userspace using MODULE_DEVICE_TABLE(), and provide it to the USB core through their usb_driver structure.

See the `usb_match_id()` function for information about how matches are performed. Briefly, you will normally use one of several macros to help construct these entries. Each entry you provide will either identify one or more specific products, or will identify a class of products which have agreed to behave the same. You should put the more specific matches towards the beginning of your table, so that driver_info can record quirks of specific products.

struct **mdio_device_id**
identifies PHY devices on an MDIO/MII bus

Definition

```
struct mdio_device_id {
  __u32 phy_id;
  __u32 phy_id_mask;
};
```

Members

phy_id The result of (mdio_read(`MII_PHYSID1`) << 16 | mdio_read(`MII_PHYSID2`)) & **phy_id_mask** for this PHY type

phy_id_mask Defines the significant bits of **phy_id**. A value of 0 is used to terminate an array of struct mdio_device_id.

struct **amba_id**
identifies a device on an AMBA bus

Definition

```
struct amba_id {
  unsigned int            id;
  unsigned int            mask;
  void *data;
};
```

Members

id The significant bits if the hardware device ID

mask Bitmask specifying which bits of the id field are significant when matching. A driver binds to a device when ((hardware device ID) & mask) == id.

data Private data used by the driver.

struct **mips_cdmm_device_id**
identifies devices in MIPS CDMM bus

Definition

```
struct mips_cdmm_device_id {
  __u8 type;
};
```

Members

type Device type identifier.

struct **mei_cl_device_id**
MEI client device identifier

Definition

```
struct mei_cl_device_id {
  char name[MEI_CL_NAME_SIZE];
  uuid_le uuid;
  __u8 version;
  kernel_ulong_t driver_info;
};
```

Members

name helper name

uuid client uuid

version client protocol version

driver_info information used by the driver.

Description

identifies mei client device by uuid and name

struct **rio_device_id**
RIO device identifier

Definition

```
struct rio_device_id {
  __u16 did, vid;
  __u16 asm_did, asm_vid;
};
```

Members

did RapidIO device ID

vid RapidIO vendor ID

asm_did RapidIO assembly device ID

asm_vid RapidIO assembly vendor ID

Description

Identifies a RapidIO device based on both the device/vendor IDs and the assembly device/vendor IDs.

struct **fsl_mc_device_id**
MC object device identifier

Definition

```
struct fsl_mc_device_id {
  __u16 vendor;
  const char obj_type[16];
};
```

Members

vendor vendor ID

obj_type MC object type

Description

Type of entries in the “device Id” table for MC object devices supported by a MC object device driver. The last entry of the table has vendor set to 0x0

struct **tb_service_id**
Thunderbolt service identifiers

Definition

```
struct tb_service_id {
  __u32 match_flags;
  char protocol_key[8 + 1];
  __u32 protocol_id;
  __u32 protocol_version;
  __u32 protocol_revision;
  kernel_ulong_t driver_data;
};
```

Members

match_flags Flags used to match the structure

protocol_key Protocol key the service supports

protocol_id Protocol id the service supports

protocol_version Version of the protocol

protocol_revision Revision of the protocol software

driver_data Driver specific data

Description

Thunderbolt XDomain services are exposed as devices where each device carries the protocol information the service supports. Thunderbolt XDomain service drivers match against that information.

struct **typec_device_id**
USB Type-C alternate mode identifiers

Definition

```
struct typec_device_id {
  __u16 svid;
  __u8 mode;
  kernel_ulong_t driver_data;
};
```

Members

svid Standard or Vendor ID

mode Mode index

driver_data Driver specific data

struct **tee_client_device_id**
tee based device identifier

Definition

```
struct tee_client_device_id {
  uuid_t uuid;
};
```

Members

uuid For TEE based client devices we use the device uuid as the identifier.

struct **wmi_device_id**
WMI device identifier

Definition

```
struct wmi_device_id {
  const char guid_string[UUID_STRING_LEN+1];
  const void *context;
};
```

Members

guid_string 36 char string of the form fa50ff2b-f2e8-45de-83fa-65417f2f49ba

context pointer to driver specific data

struct **mhi_device_id**
MHI device identification

Definition

```
struct mhi_device_id {
  const char chan[MHI_NAME_SIZE];
  kernel_ulong_t driver_data;
};
```

Members

chan MHI channel name

driver_data driver data;

2.3 Delaying, scheduling, and timer routines

struct **prev_cputime**
snapshot of system and user cputime

Definition

```
struct prev_cputime {
#ifndef CONFIG_VIRT_CPU_ACCOUNTING_NATIVE;
  u64 utime;
  u64 stime;
  raw_spinlock_t lock;
#endif;
};
```

Members

utime time spent in user mode

stime time spent in system mode

lock protects the above two fields

Description

Stores previous user/system time values such that we can guarantee monotonicity.

struct **util_est**
Estimation utilization of FAIR tasks

Definition

```
struct util_est {
  unsigned int                    enqueued;
  unsigned int                    ewma;
#define UTIL_EST_WEIGHT_SHIFT           2;
};
```

Members

enqueued instantaneous estimated utilization of a task/cpu

ewma the Exponential Weighted Moving Average (EWMA) utilization of a task

Description

Support data structure to track an Exponential Weighted Moving Average (EWMA) of a FAIR task's utilization. New samples are added to the moving average each time a task completes an activation. Sample's weight is chosen so that the EWMA will be relatively insensitive to transient changes to the task's workload.

The enqueued attribute has a slightly different meaning for tasks and cpus: - task: the task's util_avg at last task dequeue time - cfs_rq: the sum of util_est.enqueued for each RUNNABLE task on that CPU Thus, the util_est.enqueued of a task represents the contribution on the estimated utilization of the CPU where that task is currently enqueued.

Only for tasks we track a moving average of the past instantaneous estimated utilization. This allows to absorb sporadic drops in utilization of an otherwise almost periodic task.

int **pid_alive**(const struct task_struct * p)
check that a task structure is not stale

Parameters

const struct task_struct * p Task structure to be checked.

Description

Test if a process is not yet dead (at most zombie state) If pid_alive fails, then pointers within the task structure can be stale and must not be dereferenced.

Return

1 if the process is alive. 0 otherwise.

int **is_global_init**(struct task_struct * tsk)
check if a task structure is init. Since init is free to have sub-threads we need to check tgid.

Parameters

struct task_struct * tsk Task structure to be checked.

Description

Check if a task structure is the first user space task the kernel created.

Return

1 if the task structure is init. 0 otherwise.

int **task_nice**(const struct task_struct * p)
return the nice value of a given task.

Parameters

const struct task_struct * p the task in question.

Return

The nice value [-20 ⋯0 ⋯19].

bool **is_idle_task**(const struct task_struct * p)
is the specified task an idle task?

Parameters

const struct task_struct * p the task in question.

Return

1 if **p** is an idle task. 0 otherwise.

int **wake_up_process**(struct task_struct * p)
Wake up a specific process

Parameters

struct task_struct * p The process to be woken up.

Description

Attempt to wake up the nominated process and move it to the set of runnable processes.

This function executes a full memory barrier before accessing the task state.

Return

1 if the process was woken up, 0 if it was already running.

void **preempt_notifier_register**(struct preempt_notifier * notifier)
tell me when current is being preempted & rescheduled

Parameters

struct preempt_notifier * notifier notifier struct to register

void **preempt_notifier_unregister**(struct preempt_notifier * notifier)
no longer interested in preemption notifications

Parameters

struct preempt_notifier * notifier notifier struct to unregister

Description

This is not safe to call from within a preemption notifier.

__visible void notrace **preempt_schedule_notrace**(void)
preempt_schedule called by tracing

Parameters

void no arguments

Description

The tracing infrastructure uses preempt_enable_notrace to prevent recursion and tracing preempt enabling caused by the tracing infrastructure itself. But as tracing can happen in areas coming from userspace or just about to enter userspace, a preempt enable can occur before user_exit() is called. This will cause the scheduler to be called when the system is still in usermode.

To prevent this, the preempt_enable_notrace will use this function instead of preempt_schedule() to exit user context if needed before calling the scheduler.

int **sched_setscheduler**(struct task_struct * p, int policy, const struct sched_param * param)
change the scheduling policy and/or RT priority of a thread.

Parameters

struct task_struct * p the task in question.

int policy new policy.

const struct sched_param * param structure containing the new RT priority.

Return

0 on success. An error code otherwise.

Description

NOTE that the task may be already dead.

int **sched_setscheduler_nocheck**(struct task_struct * p, int policy, const struct sched_param * param)
change the scheduling policy and/or RT priority of a thread from kernelspace.

Parameters

struct task_struct * p the task in question.

int policy new policy.

const struct sched_param * param structure containing the new RT priority.

Description

Just like sched_setscheduler, only don't bother checking if the current context has permission. For example, this is needed in stop_machine(): we create temporary high priority worker threads, but our caller might not have that capability.

Return

0 on success. An error code otherwise.

void **yield**(void)
yield the current processor to other threads.

Parameters

void no arguments

Description

Do not ever use this function, there's a 99% chance you're doing it wrong.

The scheduler is at all times free to pick the calling task as the most eligible task to run, if removing the `yield()` call from your code breaks it, its already broken.

Typical broken usage is:

while (!event) `yield();`

where one assumes that `yield()` will let 'the other' process run that will make event true. If the current task is a SCHED_FIFO task that will never happen. Never use `yield()` as a progress guarantee!!

If you want to use `yield()` to wait for something, use `wait_event()`. If you want to use `yield()` to be 'nice' for others, use cond_resched(). If you still want to use `yield()`, do not!

int **yield_to**(struct task_struct * p, bool preempt)
yield the current processor to another thread in your thread group, or accelerate that thread toward the processor it's on.

Parameters

struct task_struct * p target task

bool preempt whether task preemption is allowed or not

Description

It's the caller's job to ensure that the target task struct can't go away on us before we can do any checks.

Return

true (>0) if we indeed boosted the target task. false (0) if we failed to boost the target. -ESRCH if there's no task to yield to.

int **cpupri_find_fitness**(struct cpupri * cp, struct task_struct * p, struct cpumask * lowest_mask, bool (*fitness_fn)(struct task_struct *p, int cpu))
find the best (lowest-pri) CPU in the system

Parameters

struct cpupri * cp The cpupri context

struct task_struct * p The task

struct cpumask * lowest_mask A mask to fill in with selected CPUs (or NULL)

bool (*)(struct task_struct *p, int cpu) fitness_fn A pointer to a function to do custom checks whether the CPU fits a specific criteria so that we only return those CPUs.

Note

This function returns the recommended CPUs as calculated during the current invocation. By the time the call returns, the CPUs may have in fact changed priorities any number of times. While not ideal, it is not an issue of correctness since the normal rebalancer logic will correct any discrepancies created by racing against the uncertainty of the current priority configuration.

Return

(int)bool - CPUs were found

void **cpupri_set**(struct cpupri * cp, int cpu, int newpri)
update the CPU priority setting

Parameters

struct cpupri * cp The cpupri context

int cpu The target CPU

int newpri The priority (INVALID-RT99) to assign to this CPU

Note

Assumes cpu_rq(cpu)->lock is locked

Return

(void)

int **cpupri_init**(struct cpupri * cp)
initialize the cpupri structure

Parameters

struct cpupri * cp The cpupri context

Return

-ENOMEM on memory allocation failure.

void **cpupri_cleanup**(struct cpupri * cp)
clean up the cpupri structure

Parameters

struct cpupri * cp The cpupri context

void **update_tg_load_avg**(struct cfs_rq * cfs_rq, int force)
update the tg's load avg

Parameters

struct cfs_rq * cfs_rq the cfs_rq whose avg changed

int force update regardless of how small the difference

Description

This function 'ensures': tg->load_avg := Sum tg->cfs_rq[]->avg.load. However, because tg->load_avg is a global value there are performance considerations.

In order to avoid having to look at the other cfs_rq's, we use a differential update where we store the last value we propagated. This in turn allows skipping updates if the differential is 'small'.

Updating tg's load_avg is necessary before update_cfs_share().

int **update_cfs_rq_load_avg**(u64 now, struct cfs_rq * cfs_rq)
update the cfs_rq's load/util averages

Parameters

u64 now current time, as per cfs_rq_clock_pelt()

struct cfs_rq * cfs_rq cfs_rq to update

Description

The cfs_rq avg is the direct sum of all its entities (blocked and runnable) avg. The immediate corollary is that all (fair) tasks must be attached, see post_init_entity_util_avg().

cfs_rq->avg is used for task_h_load() and update_cfs_share() for example.

Returns true if the load decayed or we removed load.

Since both these conditions indicate a changed cfs_rq->avg.load we should call `update_tg_load_avg()` when this function returns true.

void **attach_entity_load_avg**(struct cfs_rq * cfs_rq, struct sched_entity * se)

attach this entity to its cfs_rq load avg

Parameters

struct cfs_rq * cfs_rq cfs_rq to attach to

struct sched_entity * se sched_entity to attach

Description

Must call `update_cfs_rq_load_avg()` before this, since we rely on cfs_rq->avg.last_update_time being current.

void **detach_entity_load_avg**(struct cfs_rq * cfs_rq, struct sched_entity * se)

detach this entity from its cfs_rq load avg

Parameters

struct cfs_rq * cfs_rq cfs_rq to detach from

struct sched_entity * se sched_entity to detach

Description

Must call `update_cfs_rq_load_avg()` before this, since we rely on cfs_rq->avg.last_update_time being current.

unsigned long **cpu_util**(int cpu)

Parameters

int cpu the CPU to get the utilization of

Description

The unit of the return value must be the one of capacity so we can compare the utilization with the capacity of the CPU that is available for CFS task (ie cpu_capacity).

cfs_rq.avg.util_avg is the sum of running time of runnable tasks plus the recent utilization of currently non-runnable tasks on a CPU. It represents the amount of utilization of a CPU in the range [0..capacity_orig] where capacity_orig is the cpu_capacity available at the highest frequency (arch_scale_freq_capacity()). The utilization of a CPU converges towards a sum equal to or less than the current capacity (capacity_curr <= capacity_orig) of the CPU because it is the running time on this CPU scaled by capacity_curr.

The estimated utilization of a CPU is defined to be the maximum between its cfs_rq.avg.util_avg and the sum of the estimated utilization of the tasks currently RUNNABLE on that CPU. This allows to properly represent the expected utilization of a CPU which has just got a big task running since a long sleep period. At the same time however it preserves the benefits of the “blocked utilization” in describing the potential for other tasks waking up on the same CPU.

Nevertheless, cfs_rq.avg.util_avg can be higher than capacity_curr or even higher than capacity_orig because of unfortunate rounding in cfs.avg.util_avg or just after migrating tasks and new task wakeups until the average stabilizes with the new running time. We need to check that the utilization stays within the range of [0..capacity_orig] and cap it if necessary. Without utilization capping, a group could be seen as overloaded (CPU0 utilization at 121% + CPU1 utilization at 80%) whereas CPU1 has 20% of available capacity. We allow utilization to overshoot capacity_curr (but not capacity_orig) as it useful for predicting the capacity required after task migrations (scheduler-driven DVFS).

Return

the (estimated) utilization for the specified CPU

void **update_sg_lb_stats**(struct lb_env * env, struct sched_group * group, struct sg_lb_stats * sgs, int * sg_status)
Update sched_group's statistics for load balancing.

Parameters

struct lb_env * env The load balancing environment.

struct sched_group * group sched_group whose statistics are to be updated.

struct sg_lb_stats * sgs variable to hold the statistics for this group.

int * sg_status Holds flag indicating the status of the sched_group

bool **update_sd_pick_busiest**(struct lb_env * env, struct sd_lb_stats * sds, struct sched_group * sg, struct sg_lb_stats * sgs)
return 1 on busiest group

Parameters

struct lb_env * env The load balancing environment.

struct sd_lb_stats * sds sched_domain statistics

struct sched_group * sg sched_group candidate to be checked for being the busiest

struct sg_lb_stats * sgs sched_group statistics

Description

Determine if **sg** is a busier group than the previously selected busiest group.

Return

`true` if **sg** is a busier group than the previously selected busiest group. `false` otherwise.

int **idle_cpu_without**(int cpu, struct task_struct * p)
would a given CPU be idle without p ?

Parameters

int cpu the processor on which idleness is tested.

struct task_struct * p task which should be ignored.

Return

1 if the CPU would be idle. 0 otherwise.

void **update_sd_lb_stats**(struct lb_env * env, struct sd_lb_stats * sds)
Update sched_domain’s statistics for load balancing.

Parameters

struct lb_env * env The load balancing environment.

struct sd_lb_stats * sds variable to hold the statistics for this sched_domain.

void **calculate_imbalance**(struct lb_env * env, struct sd_lb_stats * sds)
Calculate the amount of imbalance present within the groups of a given sched_domain during load balance.

Parameters

struct lb_env * env load balance environment

struct sd_lb_stats * sds statistics of the sched_domain whose imbalance is to be calculated.

struct sched_group * **find_busiest_group**(struct lb_env * env)
Returns the busiest group within the sched_domain if there is an imbalance.

Parameters

struct lb_env * env The load balancing environment.

Description

Also calculates the amount of runnable load which should be moved to restore balance.

Return

- The busiest group if imbalance exists.

DECLARE_COMPLETION(work)
declare and initialize a completion structure

Parameters

work identifier for the completion structure

Description

This macro declares and initializes a completion structure. Generally used for static declarations. You should use the _ONSTACK variant for automatic variables.

DECLARE_COMPLETION_ONSTACK(work)
declare and initialize a completion structure

Parameters

work identifier for the completion structure

Description

This macro declares and initializes a completion structure on the kernel stack.

void **__init_completion**(struct completion * x)
Initialize a dynamically allocated completion

Parameters

struct completion * x pointer to completion structure that is to be initialized

Description

This inline function will initialize a dynamically created completion structure.

void **reinit_completion**(struct completion * x)
reinitialize a completion structure

Parameters

struct completion * x pointer to completion structure that is to be reinitialized

Description

This inline function should be used to reinitialize a completion structure so it can be reused. This is especially important after complete_all() is used.

unsigned long **__round_jiffies**(unsigned long j, int cpu)
function to round jiffies to a full second

Parameters

unsigned long j the time in (absolute) jiffies that should be rounded

int cpu the processor number on which the timeout will happen

Description

`__round_jiffies()` rounds an absolute time in the future (in jiffies) up or down to (approximately) full seconds. This is useful for timers for which the exact time they fire does not matter too much, as long as they fire approximately every X seconds.

By rounding these timers to whole seconds, all such timers will fire at the same time, rather than at various times spread out. The goal of this is to have the CPU wake up less, which saves power.

The exact rounding is skewed for each processor to avoid all processors firing at the exact same time, which could lead to lock contention or spurious cache line bouncing.

The return value is the rounded version of the **j** parameter.

unsigned long **__round_jiffies_relative**(unsigned long j, int cpu)
function to round jiffies to a full second

Parameters

unsigned long j the time in (relative) jiffies that should be rounded

int cpu the processor number on which the timeout will happen

Description

`__round_jiffies_relative()` rounds a time delta in the future (in jiffies) up or down to (approximately) full seconds. This is useful for timers for which the exact

time they fire does not matter too much, as long as they fire approximately every X seconds.

By rounding these timers to whole seconds, all such timers will fire at the same time, rather than at various times spread out. The goal of this is to have the CPU wake up less, which saves power.

The exact rounding is skewed for each processor to avoid all processors firing at the exact same time, which could lead to lock contention or spurious cache line bouncing.

The return value is the rounded version of the **j** parameter.

unsigned long **round_jiffies**(unsigned long j)
function to round jiffies to a full second

Parameters

unsigned long j the time in (absolute) jiffies that should be rounded

Description

round_jiffies() rounds an absolute time in the future (in jiffies) up or down to (approximately) full seconds. This is useful for timers for which the exact time they fire does not matter too much, as long as they fire approximately every X seconds.

By rounding these timers to whole seconds, all such timers will fire at the same time, rather than at various times spread out. The goal of this is to have the CPU wake up less, which saves power.

The return value is the rounded version of the **j** parameter.

unsigned long **round_jiffies_relative**(unsigned long j)
function to round jiffies to a full second

Parameters

unsigned long j the time in (relative) jiffies that should be rounded

Description

round_jiffies_relative() rounds a time delta in the future (in jiffies) up or down to (approximately) full seconds. This is useful for timers for which the exact time they fire does not matter too much, as long as they fire approximately every X seconds.

By rounding these timers to whole seconds, all such timers will fire at the same time, rather than at various times spread out. The goal of this is to have the CPU wake up less, which saves power.

The return value is the rounded version of the **j** parameter.

unsigned long **__round_jiffies_up**(unsigned long j, int cpu)
function to round jiffies up to a full second

Parameters

unsigned long j the time in (absolute) jiffies that should be rounded

int cpu the processor number on which the timeout will happen

Description

This is the same as `__round_jiffies()` except that it will never round down. This is useful for timeouts for which the exact time of firing does not matter too much, as long as they don't fire too early.

unsigned long **__round_jiffies_up_relative**(unsigned long j, int cpu)
function to round jiffies up to a full second

Parameters

unsigned long j the time in (relative) jiffies that should be rounded

int cpu the processor number on which the timeout will happen

Description

This is the same as `__round_jiffies_relative()` except that it will never round down. This is useful for timeouts for which the exact time of firing does not matter too much, as long as they don't fire too early.

unsigned long **round_jiffies_up**(unsigned long j)
function to round jiffies up to a full second

Parameters

unsigned long j the time in (absolute) jiffies that should be rounded

Description

This is the same as `round_jiffies()` except that it will never round down. This is useful for timeouts for which the exact time of firing does not matter too much, as long as they don't fire too early.

unsigned long **round_jiffies_up_relative**(unsigned long j)
function to round jiffies up to a full second

Parameters

unsigned long j the time in (relative) jiffies that should be rounded

Description

This is the same as `round_jiffies_relative()` except that it will never round down. This is useful for timeouts for which the exact time of firing does not matter too much, as long as they don't fire too early.

void **init_timer_key**(struct timer_list * timer, void (*func)(struct timer_list *), unsigned int flags, const char * name, struct lock_class_key * key)
initialize a timer

Parameters

struct timer_list * timer the timer to be initialized

void (*)(struct timer_list *) func timer callback function

unsigned int flags timer flags

const char * name name of the timer

struct lock_class_key * key lockdep class key of the fake lock used for tracking timer sync lock dependencies

Description

`init_timer_key()` must be done to a timer prior calling any of the other timer functions.

int **mod_timer_pending**(struct timer_list * timer, unsigned long expires)
modify a pending timer's timeout

Parameters

struct timer_list * timer the pending timer to be modified

unsigned long expires new timeout in jiffies

Description

`mod_timer_pending()` is the same for pending timers as `mod_timer()`, but will not re-activate and modify already deleted timers.

It is useful for unserialized use of timers.

int **mod_timer**(struct timer_list * timer, unsigned long expires)
modify a timer's timeout

Parameters

struct timer_list * timer the timer to be modified

unsigned long expires new timeout in jiffies

Description

`mod_timer()` is a more efficient way to update the expire field of an active timer (if the timer is inactive it will be activated)

mod_timer(timer, expires) is equivalent to:

del_timer(timer); timer->expires = expires; add_timer(timer);

Note that if there are multiple unserialized concurrent users of the same timer, then `mod_timer()` is the only safe way to modify the timeout, since `add_timer()` cannot modify an already running timer.

The function returns whether it has modified a pending timer or not. (ie. `mod_timer()` of an inactive timer returns 0, `mod_timer()` of an active timer returns 1.)

int **timer_reduce**(struct timer_list * timer, unsigned long expires)
Modify a timer's timeout if it would reduce the timeout

Parameters

struct timer_list * timer The timer to be modified

unsigned long expires New timeout in jiffies

Description

`timer_reduce()` is very similar to `mod_timer()`, except that it will only modify a running timer if that would reduce the expiration time (it will start a timer that isn't running).

void **add_timer**(struct timer_list * timer)
start a timer

Parameters

struct timer_list * timer the timer to be added

Description

The kernel will do a ->function(**timer**) callback from the timer interrupt at the ->expires point in the future. The current time is 'jiffies'.

The timer's ->expires, ->function fields must be set prior calling this function.

Timers with an ->expires field in the past will be executed in the next timer tick.

void **add_timer_on**(struct timer_list * timer, int cpu)
start a timer on a particular CPU

Parameters

struct timer_list * timer the timer to be added

int cpu the CPU to start it on

Description

This is not very scalable on SMP. Double adds are not possible.

int **del_timer**(struct timer_list * timer)
deactivate a timer.

Parameters

struct timer_list * timer the timer to be deactivated

Description

`del_timer()` deactivates a timer - this works on both active and inactive timers.

The function returns whether it has deactivated a pending timer or not. (ie. `del_timer()` of an inactive timer returns 0, `del_timer()` of an active timer returns 1.)

int **try_to_del_timer_sync**(struct timer_list * timer)
Try to deactivate a timer

Parameters

struct timer_list * timer timer to delete

Description

This function tries to deactivate a timer. Upon successful (ret >= 0) exit the timer is not queued and the handler is not running on any CPU.

int **del_timer_sync**(struct timer_list * timer)
deactivate a timer and wait for the handler to finish.

Parameters

struct timer_list * timer the timer to be deactivated

Description

This function only differs from `del_timer()` on SMP: besides deactivating the timer it also makes sure the handler has finished executing on other CPUs.

Synchronization rules: Callers must prevent restarting of the timer, otherwise this function is meaningless. It must not be called from interrupt contexts unless the timer is an irqsafe one. The caller must not hold locks which would prevent completion of the timer's handler. The timer's handler must not call `add_timer_on()`. Upon exit the timer is not queued and the handler is not running on any CPU.

Now `del_timer_sync()` will never return and never release somelock. The interrupt on the other CPU is waiting to grab somelock but it has interrupted the softirq that CPU0 is waiting to finish.

The function returns whether it has deactivated a pending timer or not.

Note

For !irqsafe timers, you must not hold locks that are held in interrupt context while calling this function. Even if the lock has nothing to do with the timer in question. Here's why:

```
CPU0                                CPU1
----                                ----
                                    <SOFTIRQ>
                                      call_timer_fn();
                                      base->running_timer = mytimer;
spin_lock_irq(somelock);
                                    <IRQ>
                                       spin_lock(somelock);
del_timer_sync(mytimer);
while (base->running_timer == mytimer);
```

signed long **schedule_timeout**(signed long timeout)
sleep until timeout

Parameters

signed long timeout timeout value in jiffies

Description

Make the current task sleep until **timeout** jiffies have elapsed. The function behavior depends on the current task state (see also set_current_state() description):

`TASK_RUNNING` - the scheduler is called, but the task does not sleep at all. That happens because sched_submit_work() does nothing for tasks in `TASK_RUNNING` state.

`TASK_UNINTERRUPTIBLE` - at least **timeout** jiffies are guaranteed to pass before the routine returns unless the current task is explicitly woken up, (e.g. by `wake_up_process()`).

`TASK_INTERRUPTIBLE` - the routine may return early if a signal is delivered to the current task or the current task is explicitly woken up.

The current task state is guaranteed to be `TASK_RUNNING` when this routine returns.

Specifying a **timeout** value of MAX_SCHEDULE_TIMEOUT will schedule the CPU away without a bound on the timeout. In this case the return value will be MAX_SCHEDULE_TIMEOUT.

Returns 0 when the timer has expired otherwise the remaining time in jiffies will be returned. In all cases the return value is guaranteed to be non-negative.

void **msleep**(unsigned int msecs)
sleep safely even with waitqueue interruptions

Parameters

unsigned int msecs Time in milliseconds to sleep for

unsigned long **msleep_interruptible**(unsigned int msecs)
sleep waiting for signals

Parameters

unsigned int msecs Time in milliseconds to sleep for

void **usleep_range**(unsigned long min, unsigned long max)
Sleep for an approximate time

Parameters

unsigned long min Minimum time in usecs to sleep

unsigned long max Maximum time in usecs to sleep

Description

In non-atomic context where the exact wakeup time is flexible, use usleep_range() instead of udelay(). The sleep improves responsiveness by avoiding the CPU-hogging busy-wait of udelay(), and the range reduces power usage by allowing hrtimers to take advantage of an already- scheduled interrupt instead of scheduling a new one just for this sleep.

2.4 Wait queues and Wake events

int **waitqueue_active**(struct wait_queue_head * wq_head)

- locklessly test for waiters on the queue

Parameters

struct wait_queue_head * wq_head the waitqueue to test for waiters

Description

returns true if the wait list is not empty

Use either while holding wait_queue_head::lock or when used for wakeups with an extra smp_mb() like:

```
CPU0 - waker                    CPU1 - waiter

                                for (;;) {
@cond = true;                     prepare_to_wait(&wq_head, &wait, state);
```

(continues on next page)

(continued from previous page)

```
smp_mb();                           // smp_mb() from set_current_state()
if (waitqueue_active(wq_head))           if (@cond)
  wake_up(wq_head);                        break;
                                    schedule();
                                  }
                                  finish_wait(&wq_head, &wait);
```

Because without the explicit smp_mb() it's possible for the `waitqueue_active()` load to get hoisted over the **cond** store such that we'll observe an empty wait list while the waiter might not observe **cond**.

Also note that this 'optimization' trades a spin_lock() for an smp_mb(), which (when the lock is uncontended) are of roughly equal cost.

NOTE

this function is lockless and requires care, incorrect usage _will_ lead to sporadic and non-obvious failure.

bool **wq_has_single_sleeper**(struct wait_queue_head * wq_head)
check if there is only one sleeper

Parameters

struct wait_queue_head * wq_head wait queue head

Description

Returns true of wq_head has only one sleeper on the list.

Please refer to the comment for waitqueue_active.

bool **wq_has_sleeper**(struct wait_queue_head * wq_head)
check if there are any waiting processes

Parameters

struct wait_queue_head * wq_head wait queue head

Description

Returns true if wq_head has waiting processes

Please refer to the comment for waitqueue_active.

wait_event(wq_head, condition)
sleep until a condition gets true

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

wait_event_freezable(wq_head, condition)
sleep (or freeze) until a condition gets true

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_INTERRUPTIBLE – so as not to contribute to system load) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

wait_event_timeout(wq_head, condition, timeout)
sleep until a condition gets true or a timeout elapses

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

timeout timeout, in jiffies

Description

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

Return

0 if the **condition** evaluated to `false` after the **timeout** elapsed, 1 if the **condition** evaluated to `true` after the **timeout** elapsed, or the remaining jiffies (at least 1) if the **condition** evaluated to `true` before the **timeout** elapsed.

wait_event_cmd(wq_head, condition, cmd1, cmd2)
sleep until a condition gets true

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

cmd1 the command will be executed before sleep

cmd2 the command will be executed after sleep

Description

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

wait_event_interruptible(wq_head, condition)
sleep until a condition gets true

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if **condition** evaluated to true.

wait_event_interruptible_timeout(wq_head, condition, timeout)
sleep until a condition gets true or a timeout elapses

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

timeout timeout, in jiffies

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

Return

0 if the **condition** evaluated to `false` after the **timeout** elapsed, 1 if the **condition** evaluated to `true` after the **timeout** elapsed, the remaining jiffies (at least 1) if the **condition** evaluated to `true` before the **timeout** elapsed, or -ERESTARTSYS if it was interrupted by a signal.

wait_event_hrtimeout(wq_head, condition, timeout)
sleep until a condition gets true or a timeout elapses

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

timeout timeout, as a ktime_t

Description

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

The function returns 0 if **condition** became true, or -ETIME if the timeout elapsed.

wait_event_interruptible_hrtimeout(wq, condition, timeout)
sleep until a condition gets true or a timeout elapses

Parameters

wq the waitqueue to wait on

condition a C expression for the event to wait for

timeout timeout, as a ktime_t

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

The function returns 0 if **condition** became true, -ERESTARTSYS if it was interrupted by a signal, or -ETIME if the timeout elapsed.

wait_event_idle(wq_head, condition)
wait for a condition without contributing to system load

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_IDLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

wait_event_idle_exclusive(wq_head, condition)
wait for a condition with contributing to system load

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_IDLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

The process is put on the wait queue with an WQ_FLAG_EXCLUSIVE flag set thus if other processes wait on the same list, when this process is woken further processes are not considered.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

wait_event_idle_timeout(wq_head, condition, timeout)
sleep without load until a condition becomes true or a timeout elapses

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

timeout timeout, in jiffies

Description

The process is put to sleep (TASK_IDLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

Return

0 if the **condition** evaluated to `false` after the **timeout** elapsed, 1 if the **condition** evaluated to `true` after the **timeout** elapsed, or the remaining jiffies (at least 1) if the **condition** evaluated to `true` before the **timeout** elapsed.

wait_event_idle_exclusive_timeout(wq_head, condition, timeout)
sleep without load until a condition becomes true or a timeout elapses

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

timeout timeout, in jiffies

Description

The process is put to sleep (TASK_IDLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

The process is put on the wait queue with an WQ_FLAG_EXCLUSIVE flag set thus if other processes wait on the same list, when this process is woken further processes are not considered.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

Return

0 if the **condition** evaluated to `false` after the **timeout** elapsed, 1 if the **condition** evaluated to `true` after the **timeout** elapsed, or the remaining jiffies (at least 1) if the **condition** evaluated to `true` before the **timeout** elapsed.

wait_event_interruptible_locked(wq, condition)
sleep until a condition gets true

Parameters

wq the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq** is woken up.

It must be called with wq.lock being held. This spinlock is unlocked while sleeping but **condition** testing is done while lock is held and when this macro exits the lock is held.

The lock is locked/unlocked using spin_lock()/spin_unlock() functions which must match the way they are locked/unlocked outside of this macro.

wake_up_locked() has to be called after changing any variable that could change the result of the wait condition.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if **condition** evaluated to true.

wait_event_interruptible_locked_irq(wq, condition)
sleep until a condition gets true

Parameters

wq the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq** is woken up.

It must be called with wq.lock being held. This spinlock is unlocked while sleeping but **condition** testing is done while lock is held and when this macro exits the lock is held.

The lock is locked/unlocked using spin_lock_irq()/spin_unlock_irq() functions which must match the way they are locked/unlocked outside of this macro.

wake_up_locked() has to be called after changing any variable that could change the result of the wait condition.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if **condition** evaluated to true.

wait_event_interruptible_exclusive_locked(wq, condition)
sleep exclusively until a condition gets true

Parameters

wq the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq** is woken up.

It must be called with wq.lock being held. This spinlock is unlocked while sleeping but **condition** testing is done while lock is held and when this macro exits the lock is held.

The lock is locked/unlocked using spin_lock()/spin_unlock() functions which must match the way they are locked/unlocked outside of this macro.

The process is put on the wait queue with an WQ_FLAG_EXCLUSIVE flag set thus when other process waits process on the list if this process is awaken further processes are not considered.

wake_up_locked() has to be called after changing any variable that could change the result of the wait condition.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if **condition** evaluated to true.

wait_event_interruptible_exclusive_locked_irq(wq, condition)
sleep until a condition gets true

Parameters

wq the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq** is woken up.

It must be called with wq.lock being held. This spinlock is unlocked while sleeping but **condition** testing is done while lock is held and when this macro exits the lock is held.

The lock is locked/unlocked using spin_lock_irq()/spin_unlock_irq() functions which must match the way they are locked/unlocked outside of this macro.

The process is put on the wait queue with an WQ_FLAG_EXCLUSIVE flag set thus when other process waits process on the list if this process is awaken further processes are not considered.

wake_up_locked() has to be called after changing any variable that could change the result of the wait condition.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if **condition** evaluated to true.

wait_event_killable(wq_head, condition)
sleep until a condition gets true

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

Description

The process is put to sleep (TASK_KILLABLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if **condition** evaluated to true.

wait_event_killable_timeout(wq_head, condition, timeout)
sleep until a condition gets true or a timeout elapses

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

timeout timeout, in jiffies

Description

The process is put to sleep (TASK_KILLABLE) until the **condition** evaluates to true or a kill signal is received. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

Only kill signals interrupt this process.

Return

0 if the **condition** evaluated to `false` after the **timeout** elapsed, 1 if the **condition** evaluated to `true` after the **timeout** elapsed, the remaining jiffies (at least 1) if the **condition** evaluated to `true` before the **timeout** elapsed, or `-ERESTARTSYS` if it was interrupted by a kill signal.

wait_event_lock_irq_cmd(wq_head, condition, lock, cmd)
sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

lock a locked spinlock_t, which will be released before cmd and schedule() and reacquired afterwards.

cmd a command which is invoked outside the critical section before sleep

Description

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before invoking the cmd and going to sleep and is reacquired afterwards.

wait_event_lock_irq(wq_head, condition, lock)
sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

lock a locked spinlock_t, which will be released before schedule() and reacquired afterwards.

Description

The process is put to sleep (TASK_UNINTERRUPTIBLE) until the **condition** evaluates to true. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before going to sleep and is reacquired afterwards.

wait_event_interruptible_lock_irq_cmd(wq_head, condition, lock, cmd)
sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

lock a locked spinlock_t, which will be released before cmd and schedule() and reacquired afterwards.

cmd a command which is invoked outside the critical section before sleep

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or a signal is received. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before invoking the cmd and going to sleep and is reacquired afterwards.

The macro will return -ERESTARTSYS if it was interrupted by a signal and 0 if **condition** evaluated to true.

wait_event_interruptible_lock_irq(wq_head, condition, lock)
sleep until a condition gets true. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

lock a locked spinlock_t, which will be released before schedule() and reacquired afterwards.

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or signal is received. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before going to sleep and is reacquired afterwards.

The macro will return -ERESTARTSYS if it was interrupted by a signal and 0 if **condition** evaluated to true.

wait_event_interruptible_lock_irq_timeout(wq_head, condition, lock, timeout)
sleep until a condition gets true or a timeout elapses. The condition is checked under the lock. This is expected to be called with the lock taken.

Parameters

wq_head the waitqueue to wait on

condition a C expression for the event to wait for

lock a locked spinlock_t, which will be released before schedule() and reacquired afterwards.

timeout timeout, in jiffies

Description

The process is put to sleep (TASK_INTERRUPTIBLE) until the **condition** evaluates to true or signal is received. The **condition** is checked each time the waitqueue **wq_head** is woken up.

wake_up() has to be called after changing any variable that could change the result of the wait condition.

This is supposed to be called while holding the lock. The lock is dropped before going to sleep and is reacquired afterwards.

The function returns 0 if the **timeout** elapsed, -ERESTARTSYS if it was interrupted by a signal, and the remaining jiffies otherwise if the condition evaluated to true before the timeout elapsed.

void **__wake_up**(struct wait_queue_head * wq_head, unsigned int mode, int nr_exclusive, void * key)
wake up threads blocked on a waitqueue.

Parameters

struct wait_queue_head * wq_head the waitqueue

unsigned int mode which threads

int nr_exclusive how many wake-one or wake-many threads to wake up

void * key is directly passed to the wakeup function

Description

If this function wakes up a task, it executes a full memory barrier before accessing the task state.

void **__wake_up_sync_key**(struct wait_queue_head * wq_head, unsigned int mode, void * key)
wake up threads blocked on a waitqueue.

Parameters

struct wait_queue_head * wq_head the waitqueue

unsigned int mode which threads

void * key opaque value to be passed to wakeup targets

Description

The sync wakeup differs that the waker knows that it will schedule away soon, so while the target thread will be woken up, it will not be migrated to another CPU - ie. the two threads are 'synchronized' with each other. This can prevent needless bouncing between CPUs.

On UP it can prevent extra preemption.

If this function wakes up a task, it executes a full memory barrier before accessing the task state.

void **__wake_up_locked_sync_key**(struct wait_queue_head * wq_head, unsigned int mode, void * key)
wake up a thread blocked on a locked waitqueue.

Parameters

struct wait_queue_head * wq_head the waitqueue

unsigned int mode which threads

void * key opaque value to be passed to wakeup targets

Description

The sync wakeup differs in that the waker knows that it will schedule away soon, so while the target thread will be woken up, it will not be migrated to another CPU - ie. the two threads are 'synchronized' with each other. This can prevent needless bouncing between CPUs.

On UP it can prevent extra preemption.

If this function wakes up a task, it executes a full memory barrier before accessing the task state.

void **finish_wait**(struct wait_queue_head * wq_head, struct wait_queue_entry * wq_entry)
clean up after waiting in a queue

Parameters

struct wait_queue_head * wq_head waitqueue waited on

struct wait_queue_entry * wq_entry wait descriptor

Description

Sets current thread back to running state and removes the wait descriptor from the given waitqueue if still queued.

2.5 High-resolution timers

ktime_t **ktime_set**(const s64 secs, const unsigned long nsecs)
Set a ktime_t variable from a seconds/nanoseconds value

Parameters

const s64 secs seconds to set

const unsigned long nsecs nanoseconds to set

Return

The ktime_t representation of the value.

int **ktime_compare**(const ktime_t cmp1, const ktime_t cmp2)
Compares two ktime_t variables for less, greater or equal

Parameters

const ktime_t cmp1 comparable1

const ktime_t cmp2 comparable2

Return

… cmp1 < cmp2: return <0 cmp1 == cmp2: return 0 cmp1 > cmp2: return >0

bool **ktime_after**(const ktime_t cmp1, const ktime_t cmp2)
Compare if a ktime_t value is bigger than another one.

Parameters

const ktime_t cmp1 comparable1

const ktime_t cmp2 comparable2

Return

true if cmp1 happened after cmp2.

bool **ktime_before**(const ktime_t cmp1, const ktime_t cmp2)
Compare if a ktime_t value is smaller than another one.

Parameters

const ktime_t cmp1 comparable1

const ktime_t cmp2 comparable2

Return

true if cmp1 happened before cmp2.

bool **ktime_to_timespec64_cond**(const ktime_t kt, struct timespec64 * ts)
convert a ktime_t variable to timespec64 format only if the variable contains data

Parameters

const ktime_t kt the ktime_t variable to convert

struct timespec64 * ts the timespec variable to store the result in

Return

`true` if there was a successful conversion, `false` if kt was 0.

struct **hrtimer**
the basic hrtimer structure

Definition

```
struct hrtimer {
  struct timerqueue_node          node;
  ktime_t _softexpires;
  enum hrtimer_restart            (*function)(struct hrtimer *);
  struct hrtimer_clock_base       *base;
  u8 state;
  u8 is_rel;
  u8 is_soft;
  u8 is_hard;
};
```

Members

node timerqueue node, which also manages node.expires, the absolute expiry time in the hrtimers internal representation. The time is related to the clock on which the timer is based. Is setup by adding slack to the _softexpires value. For non range timers identical to _softexpires.

_softexpires the absolute earliest expiry time of the hrtimer. The time which was given as expiry time when the timer was armed.

function timer expiry callback function

base pointer to the timer base (per cpu and per clock)

state state information (See bit values above)

is_rel Set if the timer was armed relative

is_soft Set if hrtimer will be expired in soft interrupt context.

is_hard Set if hrtimer will be expired in hard interrupt context even on RT.

Description

The hrtimer structure must be initialized by `hrtimer_init()`

struct **hrtimer_sleeper**
simple sleeper structure

Definition

```
struct hrtimer_sleeper {
  struct hrtimer timer;
  struct task_struct *task;
};
```

Members

timer embedded timer structure

task task to wake up

Description

task is set to NULL, when the timer expires.

struct **hrtimer_clock_base**
the timer base for a specific clock

Definition

```
struct hrtimer_clock_base {
  struct hrtimer_cpu_base *cpu_base;
  unsigned int            index;
  clockid_t clockid;
  seqcount_t seq;
  struct hrtimer          *running;
  struct timerqueue_head  active;
  ktime_t (*get_time)(void);
  ktime_t offset;
};
```

Members

cpu_base per cpu clock base

index clock type index for per_cpu support when moving a timer to a base on another cpu.

clockid clock id for per_cpu support

seq seqcount around __run_hrtimer

running pointer to the currently running hrtimer

active red black tree root node for the active timers

get_time function to retrieve the current time of the clock

offset offset of this clock to the monotonic base

struct **hrtimer_cpu_base**
the per cpu clock bases

Definition

```
struct hrtimer_cpu_base {
  raw_spinlock_t lock;
  unsigned int                    cpu;
  unsigned int                    active_bases;
  unsigned int                    clock_was_set_seq;
  unsigned int                    hres_active             : 1,in_hrtirq
↪           : 1,hang_detected          : 1, softirq_activated       : 1;
#ifdef CONFIG_HIGH_RES_TIMERS;
  unsigned int                    nr_events;
  unsigned short                  nr_retries;
  unsigned short                  nr_hangs;
  unsigned int                    max_hang_time;
#endif;
#ifdef CONFIG_PREEMPT_RT;
  spinlock_t softirq_expiry_lock;
  atomic_t timer_waiters;
#endif;
  ktime_t expires_next;
  struct hrtimer                  *next_timer;
  ktime_t softirq_expires_next;
  struct hrtimer                  *softirq_next_timer;
  struct hrtimer_clock_base       clock_base[HRTIMER_MAX_CLOCK_BASES];
};
```

Members

lock lock protecting the base and associated clock bases and timers

cpu cpu number

active_bases Bitfield to mark bases with active timers

clock_was_set_seq Sequence counter of clock was set events

hres_active State of high resolution mode

in_hrtirq hrtimer_interrupt() is currently executing

hang_detected The last hrtimer interrupt detected a hang

softirq_activated displays, if the softirq is raised - update of softirq related settings is not required then.

nr_events Total number of hrtimer interrupt events

nr_retries Total number of hrtimer interrupt retries

nr_hangs Total number of hrtimer interrupt hangs

max_hang_time Maximum time spent in hrtimer_interrupt

softirq_expiry_lock Lock which is taken while softirq based hrtimer are expired

timer_waiters A `hrtimer_cancel()` invocation waits for the timer callback to finish.

expires_next absolute time of the next event, is required for remote hrtimer enqueue; it is the total first expiry time (hard and soft hrtimer are taken into account)

next_timer Pointer to the first expiring timer

softirq_expires_next Time to check, if soft queues needs also to be expired

softirq_next_timer Pointer to the first expiring softirq based timer

clock_base array of clock bases for this cpu

Note

next_timer is just an optimization for __remove_hrtimer(). Do not dereference the pointer because it is not reliable on cross cpu removals.

void **hrtimer_start**(struct hrtimer * timer, ktime_t tim, const enum hrtimer_mode mode)
(re)start an hrtimer

Parameters

struct hrtimer * timer the timer to be added

ktime_t tim expiry time

const enum hrtimer_mode mode timer mode: absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL), and pinned (HRTIMER_MODE_PINNED); softirq based mode is considered for debug purpose only!

bool **hrtimer_is_queued**(struct hrtimer * timer)

Parameters

struct hrtimer * timer Timer to check

Return

True if the timer is queued, false otherwise

Description

The function can be used lockless, but it gives only a current snapshot.

u64 **hrtimer_forward_now**(struct hrtimer * timer, ktime_t interval)
forward the timer expiry so it expires after now

Parameters

struct hrtimer * timer hrtimer to forward

ktime_t interval the interval to forward

Description

Forward the timer expiry so it will expire after the current time of the hrtimer clock base. Returns the number of overruns.

Can be safely called from the callback function of **timer**. If called from other contexts **timer** must neither be enqueued nor running the callback and the caller needs to take care of serialization.

Note

This only updates the timer expiry value and does not requeue the timer.

u64 **hrtimer_forward**(struct hrtimer * timer, ktime_t now, ktime_t interval)
forward the timer expiry

Parameters

struct hrtimer * timer hrtimer to forward

ktime_t now forward past this time

ktime_t interval the interval to forward

Description

Forward the timer expiry so it will expire in the future. Returns the number of overruns.

Can be safely called from the callback function of **timer**. If called from other contexts **timer** must neither be enqueued nor running the callback and the caller needs to take care of serialization.

Note

This only updates the timer expiry value and does not requeue the timer.

void **hrtimer_start_range_ns**(struct hrtimer * timer, ktime_t tim, u64 delta_ns, const enum hrtimer_mode mode)
(re)start an hrtimer

Parameters

struct hrtimer * timer the timer to be added

ktime_t tim expiry time

u64 delta_ns “slack” range for the timer

const enum hrtimer_mode mode timer mode: absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL), and pinned (HRTIMER_MODE_PINNED); softirq based mode is considered for debug purpose only!

int **hrtimer_try_to_cancel**(struct hrtimer * timer)
try to deactivate a timer

Parameters

struct hrtimer * timer hrtimer to stop

Return

- 0 when the timer was not active
- 1 when the timer was active
- -1 when the timer is currently executing the callback function and cannot be stopped

int **hrtimer_cancel**(struct hrtimer * timer)
cancel a timer and wait for the handler to finish.

Parameters

struct hrtimer * timer the timer to be cancelled

Return

0 when the timer was not active 1 when the timer was active

ktime_t **__hrtimer_get_remaining**(const struct hrtimer * timer, bool adjust)
get remaining time for the timer

Parameters

const struct hrtimer * timer the timer to read

bool adjust adjust relative timers when CONFIG_TIME_LOW_RES=y

void **hrtimer_init**(struct hrtimer * timer, clockid_t clock_id, enum hrtimer_mode mode)
initialize a timer to the given clock

Parameters

struct hrtimer * timer the timer to be initialized

clockid_t clock_id the clock to be used

enum hrtimer_mode mode The modes which are relevant for initialization: HRTIMER_MODE_ABS, HRTIMER_MODE_REL, HRTIMER_MODE_ABS_SOFT, HRTIMER_MODE_REL_SOFT

The PINNED variants of the above can be handed in, but the PINNED bit is ignored as pinning happens when the hrtimer is started

void **hrtimer_sleeper_start_expires**(struct hrtimer_sleeper * sl, enum hrtimer_mode mode)
Start a hrtimer sleeper timer

Parameters

struct hrtimer_sleeper * sl sleeper to be started

enum hrtimer_mode mode timer mode abs/rel

Description

Wrapper around hrtimer_start_expires() for hrtimer_sleeper based timers to allow PREEMPT_RT to tweak the delivery mode (soft/hardirq context)

void **hrtimer_init_sleeper**(struct hrtimer_sleeper * sl, clockid_t clock_id, enum hrtimer_mode mode)
initialize sleeper to the given clock

Parameters

struct hrtimer_sleeper * sl sleeper to be initialized

clockid_t clock_id the clock to be used

enum hrtimer_mode mode timer mode abs/rel

int **schedule_hrtimeout_range**(ktime_t * expires, u64 delta, const enum hrtimer_mode mode)
sleep until timeout

Parameters

ktime_t * expires timeout value (ktime_t)

u64 delta slack in expires timeout (ktime_t)

const enum hrtimer_mode mode timer mode

Description

Make the current task sleep until the given expiry time has elapsed. The routine will return immediately unless the current task state has been set (see set_current_state()).

The **delta** argument gives the kernel the freedom to schedule the actual wakeup to a time that is both power and performance friendly. The kernel give the normal best effort behavior for "**expires**+**delta**", but may decide to fire the timer earlier, but no earlier than **expires**.

You can set the task state as follows -

TASK_UNINTERRUPTIBLE - at least **timeout** time is guaranteed to pass before the routine returns unless the current task is explicitly woken up, (e.g. by wake_up_process()).

TASK_INTERRUPTIBLE - the routine may return early if a signal is delivered to the current task or the current task is explicitly woken up.

The current task state is guaranteed to be TASK_RUNNING when this routine returns.

Returns 0 when the timer has expired. If the task was woken before the timer expired by a signal (only possible in state TASK_INTERRUPTIBLE) or by an explicit wakeup, it returns -EINTR.

int **schedule_hrtimeout**(ktime_t * expires, const enum hrtimer_mode mode)

sleep until timeout

Parameters

ktime_t * expires timeout value (ktime_t)

const enum hrtimer_mode mode timer mode

Description

Make the current task sleep until the given expiry time has elapsed. The routine will return immediately unless the current task state has been set (see set_current_state()).

You can set the task state as follows -

TASK_UNINTERRUPTIBLE - at least **timeout** time is guaranteed to pass before the routine returns unless the current task is explicitly woken up, (e.g. by wake_up_process()).

TASK_INTERRUPTIBLE - the routine may return early if a signal is delivered to the current task or the current task is explicitly woken up.

The current task state is guaranteed to be TASK_RUNNING when this routine returns.

Returns 0 when the timer has expired. If the task was woken before the timer expired by a signal (only possible in state TASK_INTERRUPTIBLE) or by an explicit

wakeup, it returns -EINTR.

2.6 Workqueues and Kevents

struct **workqueue_attrs**
A struct for workqueue attributes.

Definition

```
struct workqueue_attrs {
  int nice;
  cpumask_var_t cpumask;
  bool no_numa;
};
```

Members

nice nice level

cpumask allowed CPUs

no_numa disable NUMA affinity

Unlike other fields, `no_numa` isn't a property of a worker_pool. It only modifies how `apply_workqueue_attrs()` select pools and thus doesn't participate in pool hash calculations or equality comparisons.

Description

This can be used to change attributes of an unbound workqueue.

work_pending(work)
Find out whether a work item is currently pending

Parameters

work The work item in question

delayed_work_pending(w)
Find out whether a delayable work item is currently pending

Parameters

w The work item in question

struct workqueue_struct * **alloc_workqueue**(const char * fmt, unsigned int flags, int max_active, ...)
allocate a workqueue

Parameters

const char * fmt printf format for the name of the workqueue

unsigned int flags WQ_* flags

int max_active max in-flight work items, 0 for default remaining args: args for **fmt**

... variable arguments

Description

Allocate a workqueue with the specified parameters. For detailed information on WQ_* flags, please refer to Documentation/core-api/workqueue.rst.

Return

Pointer to the allocated workqueue on success, `NULL` on failure.

alloc_ordered_workqueue(fmt, flags, args)
allocate an ordered workqueue

Parameters

fmt printf format for the name of the workqueue

flags WQ_* flags (only WQ_FREEZABLE and WQ_MEM_RECLAIM are meaningful)

args args for **fmt**

Description

Allocate an ordered workqueue. An ordered workqueue executes at most one work item at any given time in the queued order. They are implemented as unbound workqueues with **max_active** of one.

Return

Pointer to the allocated workqueue on success, `NULL` on failure.

bool **queue_work**(struct workqueue_struct * wq, struct work_struct * work)
queue work on a workqueue

Parameters

struct workqueue_struct * wq workqueue to use

struct work_struct * work work to queue

Description

Returns `false` if **work** was already on a queue, `true` otherwise.

We queue the work to the CPU on which it was submitted, but if the CPU dies it can be processed by another CPU.

Memory-ordering properties: If it returns `true`, guarantees that all stores preceding the call to queue_work() in the program order will be visible from the CPU which will execute **work** by the time such work executes, e.g.,

{ x is initially 0 }

CPU0 CPU1

WRITE_ONCE(x, 1); [**work** is being executed] r0 = queue_work(wq, work); r1 = READ_ONCE(x);

Forbids: r0 == true && r1 == 0

bool **queue_delayed_work**(struct workqueue_struct * wq, struct delayed_work * dwork, unsigned long delay)
queue work on a workqueue after delay

Parameters

struct workqueue_struct * wq workqueue to use

struct delayed_work * dwork delayable work to queue

unsigned long delay number of jiffies to wait before queueing

Description

Equivalent to `queue_delayed_work_on()` but tries to use the local CPU.

bool **mod_delayed_work**(struct workqueue_struct * wq, struct delayed_work * dwork, unsigned long delay)
modify delay of or queue a delayed work

Parameters

struct workqueue_struct * wq workqueue to use

struct delayed_work * dwork work to queue

unsigned long delay number of jiffies to wait before queueing

Description

`mod_delayed_work_on()` on local CPU.

bool **schedule_work_on**(int cpu, struct work_struct * work)
put work task on a specific cpu

Parameters

int cpu cpu to put the work task on

struct work_struct * work job to be done

Description

This puts a job on a specific cpu

bool **schedule_work**(struct work_struct * work)
put work task in global workqueue

Parameters

struct work_struct * work job to be done

Description

Returns `false` if **work** was already on the kernel-global workqueue and `true` otherwise.

This puts a job in the kernel-global workqueue if it was not already queued and leaves it in the same position on the kernel-global workqueue otherwise.

Shares the same memory-ordering properties of queue_work(), cf. the DocBook header of queue_work().

void **flush_scheduled_work**(void)
ensure that any scheduled work has run to completion.

Parameters

void no arguments

Description

Forces execution of the kernel-global workqueue and blocks until its completion.

Think twice before calling this function! It's very easy to get into trouble if you don't take great care. Either of the following situations will lead to deadlock:

> One of the work items currently on the workqueue needs to acquire a lock held by your code or its caller.
>
> Your code is running in the context of a work routine.

They will be detected by lockdep when they occur, but the first might not occur very often. It depends on what work items are on the workqueue and what locks they need, which you have no control over.

In most situations flushing the entire workqueue is overkill; you merely need to know that a particular work item isn't queued and isn't running. In such cases you should use `cancel_delayed_work_sync()` or `cancel_work_sync()` instead.

bool **schedule_delayed_work_on**(int cpu, struct delayed_work * dwork, unsigned long delay)
queue work in global workqueue on CPU after delay

Parameters

int cpu cpu to use

struct delayed_work * dwork job to be done

unsigned long delay number of jiffies to wait

Description

After waiting for a given time this puts a job in the kernel-global workqueue on the specified CPU.

bool **schedule_delayed_work**(struct delayed_work * dwork, unsigned long delay)
put work task in global workqueue after delay

Parameters

struct delayed_work * dwork job to be done

unsigned long delay number of jiffies to wait or 0 for immediate execution

Description

After waiting for a given time this puts a job in the kernel-global workqueue.

bool **queue_work_on**(int cpu, struct workqueue_struct * wq, struct work_struct * work)
queue work on specific cpu

Parameters

int cpu CPU number to execute work on

struct workqueue_struct * wq workqueue to use

struct work_struct * work work to queue

Description

We queue the work to a specific CPU, the caller must ensure it can't go away.

Return

`false` if **work** was already on a queue, `true` otherwise.

bool **queue_work_node**(int node, struct workqueue_struct * wq, struct work_struct * work)
queue work on a "random" cpu for a given NUMA node

Parameters

int node NUMA node that we are targeting the work for

struct workqueue_struct * wq workqueue to use

struct work_struct * work work to queue

Description

We queue the work to a "random" CPU within a given NUMA node. The basic idea here is to provide a way to somehow associate work with a given NUMA node.

This function will only make a best effort attempt at getting this onto the right NUMA node. If no node is requested or the requested node is offline then we just fall back to standard queue_work behavior.

Currently the "random" CPU ends up being the first available CPU in the intersection of cpu_online_mask and the cpumask of the node, unless we are running on the node. In that case we just use the current CPU.

Return

`false` if **work** was already on a queue, `true` otherwise.

bool **queue_delayed_work_on**(int cpu, struct workqueue_struct * wq, struct delayed_work * dwork, unsigned long delay)
queue work on specific CPU after delay

Parameters

int cpu CPU number to execute work on

struct workqueue_struct * wq workqueue to use

struct delayed_work * dwork work to queue

unsigned long delay number of jiffies to wait before queueing

Return

`false` if **work** was already on a queue, `true` otherwise. If **delay** is zero and **dwork** is idle, it will be scheduled for immediate execution.

bool **mod_delayed_work_on**(int cpu, struct workqueue_struct * wq, struct delayed_work * dwork, unsigned long delay)
modify delay of or queue a delayed work on specific CPU

Parameters

int cpu CPU number to execute work on

struct workqueue_struct * wq workqueue to use

struct delayed_work * dwork work to queue

unsigned long delay number of jiffies to wait before queueing

Description

If **dwork** is idle, equivalent to `queue_delayed_work_on();` otherwise, modify **dwork**' s timer so that it expires after **delay**. If **delay** is zero, **work** is guaranteed to be scheduled immediately regardless of its current state.

This function is safe to call from any context including IRQ handler. See try_to_grab_pending() for details.

Return

`false` if **dwork** was idle and queued, `true` if **dwork** was pending and its timer was modified.

bool **queue_rcu_work**(struct workqueue_struct * wq, struct rcu_work * rwork)
queue work after a RCU grace period

Parameters

struct workqueue_struct * wq workqueue to use

struct rcu_work * rwork work to queue

Return

`false` if **rwork** was already pending, `true` otherwise. Note that a full RCU grace period is guaranteed only after a `true` return. While **rwork** is guaranteed to be executed after a `false` return, the execution may happen before a full RCU grace period has passed.

void **flush_workqueue**(struct workqueue_struct * wq)
ensure that any scheduled work has run to completion.

Parameters

struct workqueue_struct * wq workqueue to flush

Description

This function sleeps until all work items which were queued on entry have finished execution, but it is not livelocked by new incoming ones.

void **drain_workqueue**(struct workqueue_struct * wq)
drain a workqueue

Parameters

struct workqueue_struct * wq workqueue to drain

Description

Wait until the workqueue becomes empty. While draining is in progress, only chain queueing is allowed. IOW, only currently pending or running work items on **wq** can queue further work items on it. **wq** is flushed repeatedly until it becomes empty. The number of flushing is determined by the depth of chaining and should be relatively short. Whine if it takes too long.

bool **flush_work**(struct work_struct * work)
wait for a work to finish executing the last queueing instance

Parameters

struct work_struct * work the work to flush

Description

Wait until **work** has finished execution. **work** is guaranteed to be idle on return if it hasn't been requeued since flush started.

Return

`true` if `flush_work()` waited for the work to finish execution, `false` if it was already idle.

bool **cancel_work_sync**(struct work_struct * work)
cancel a work and wait for it to finish

Parameters

struct work_struct * work the work to cancel

Description

Cancel **work** and wait for its execution to finish. This function can be used even if the work re-queues itself or migrates to another workqueue. On return from this function, **work** is guaranteed to be not pending or executing on any CPU.

cancel_work_sync(`delayed_work->work`) must not be used for delayed_work's. Use `cancel_delayed_work_sync()` instead.

The caller must ensure that the workqueue on which **work** was last queued can't be destroyed before this function returns.

Return

`true` if **work** was pending, `false` otherwise.

bool **flush_delayed_work**(struct delayed_work * dwork)
wait for a dwork to finish executing the last queueing

Parameters

struct delayed_work * dwork the delayed work to flush

Description

Delayed timer is cancelled and the pending work is queued for immediate execution. Like `flush_work()`, this function only considers the last queueing instance of **dwork**.

Return

`true` if `flush_work()` waited for the work to finish execution, `false` if it was already idle.

bool **flush_rcu_work**(struct rcu_work * rwork)
wait for a rwork to finish executing the last queueing

Parameters

struct rcu_work * rwork the rcu work to flush

Return

`true` if `flush_rcu_work()` waited for the work to finish execution, `false` if it was already idle.

bool **cancel_delayed_work**(struct delayed_work * dwork)
cancel a delayed work

Parameters

struct delayed_work * dwork delayed_work to cancel

Description

Kill off a pending delayed_work.

This function is safe to call from any context including IRQ handler.

Return

`true` if **dwork** was pending and canceled; `false` if it wasn't pending.

Note

The work callback function may still be running on return, unless it returns `true` and the work doesn't re-arm itself. Explicitly flush or use `cancel_delayed_work_sync()` to wait on it.

bool **cancel_delayed_work_sync**(struct delayed_work * dwork)
cancel a delayed work and wait for it to finish

Parameters

struct delayed_work * dwork the delayed work cancel

Description

This is `cancel_work_sync()` for delayed works.

Return

`true` if **dwork** was pending, `false` otherwise.

int **execute_in_process_context**(work_func_t fn, struct execute_work * ew)
reliably execute the routine with user context

Parameters

work_func_t fn the function to execute

struct execute_work * ew guaranteed storage for the execute work structure (must be available when the work executes)

Description

Executes the function immediately if process context is available, otherwise schedules the function for delayed execution.

Return

0 - function was executed 1 - function was scheduled for execution

void **destroy_workqueue**(struct workqueue_struct * wq)
safely terminate a workqueue

Parameters

struct workqueue_struct * wq target workqueue

Description

Safely destroy a workqueue. All work currently pending will be done first.

void **workqueue_set_max_active**(struct workqueue_struct * wq, int max_active)
adjust max_active of a workqueue

Parameters

struct workqueue_struct * wq target workqueue

int max_active new max_active value.

Description

Set max_active of **wq** to **max_active**.

Context

Don't call from IRQ context.

struct work_struct * **current_work**(void)
retrieve `current` task's work struct

Parameters

void no arguments

Description

Determine if `current` task is a workqueue worker and what it's working on. Useful to find out the context that the `current` task is running in.

Return

work struct if `current` task is a workqueue worker, `NULL` otherwise.

bool **workqueue_congested**(int cpu, struct workqueue_struct * wq)
test whether a workqueue is congested

Parameters

int cpu CPU in question

struct workqueue_struct * wq target workqueue

Description

Test whether **wq**'s cpu workqueue for **cpu** is congested. There is no synchronization around this function and the test result is unreliable and only useful as advisory hints or for debugging.

If **cpu** is WORK_CPU_UNBOUND, the test is performed on the local CPU. Note that both per-cpu and unbound workqueues may be associated with multiple pool_workqueues which have separate congested states. A workqueue being congested on one CPU doesn't mean the workqueue is also contested on other CPUs / NUMA nodes.

Return

`true` if congested, `false` otherwise.

unsigned int **work_busy**(struct work_struct * work)
test whether a work is currently pending or running

Parameters

struct work_struct * work the work to be tested

Description

Test whether **work** is currently pending or running. There is no synchronization around this function and the test result is unreliable and only useful as advisory hints or for debugging.

Return

OR'd bitmask of WORK_BUSY_* bits.

void **set_worker_desc**(const char * fmt, ...)
set description for the current work item

Parameters

const char * fmt printf-style format string

... arguments for the format string

Description

This function can be called by a running work function to describe what the work item is about. If the worker task gets dumped, this information will be printed out together to help debugging. The description can be at most WORKER_DESC_LEN including the trailing '0'.

long **work_on_cpu**(int cpu, long (*fn)(void *), void * arg)
run a function in thread context on a particular cpu

Parameters

int cpu the cpu to run on

long (*)(void *) fn the function to run

void * arg the function arg

Description

It is up to the caller to ensure that the cpu doesn't go offline. The caller must not hold any locks which would prevent **fn** from completing.

Return

The value **fn** returns.

long **work_on_cpu_safe**(int cpu, long (*fn)(void *), void * arg)
run a function in thread context on a particular cpu

Parameters

int cpu the cpu to run on

long (*)(void *) fn the function to run

void * arg the function argument

Description

Disables CPU hotplug and calls `work_on_cpu()`. The caller must not hold any locks which would prevent **fn** from completing.

Return

The value **fn** returns.

2.7 Internal Functions

int **wait_task_stopped**(struct wait_opts * wo, int ptrace, struct task_struct * p)
Wait for `TASK_STOPPED` or `TASK_TRACED`

Parameters

struct wait_opts * wo wait options

int ptrace is the wait for ptrace

struct task_struct * p task to wait for

Description

Handle sys_wait4() work for `p` in state `TASK_STOPPED` or `TASK_TRACED`.

Context

read_lock(`tasklist_lock`), which is released if return value is non-zero. Also, grabs and releases **p->sighand->siglock**.

Return

0 if wait condition didn't exist and search for other wait conditions should continue. Non-zero return, -errno on failure and **p**'s pid on success, implies that tasklist_lock is released and wait condition search should terminate.

bool **task_set_jobctl_pending**(struct task_struct * task, unsigned long mask)
set jobctl pending bits

Parameters

struct task_struct * task target task

unsigned long mask pending bits to set

Description

Clear **mask** from **task->jobctl**. **mask** must be subset of `JOBCTL_PENDING_MASK` | `JOBCTL_STOP_CONSUME` | `JOBCTL_STOP_SIGMASK` | `JOBCTL_TRAPPING`. If stop signo is being set, the existing signo is cleared. If **task** is already being killed or exiting, this function becomes noop.

Context

Must be called with **task->sighand->siglock** held.

Return

`true` if **mask** is set, `false` if made noop because **task** was dying.

void **task_clear_jobctl_trapping**(struct task_struct * task)
clear jobctl trapping bit

Parameters

struct task_struct * task target task

Description

If JOBCTL_TRAPPING is set, a ptracer is waiting for us to enter TRACED. Clear it and wake up the ptracer. Note that we don't need any further locking. **task->siglock** guarantees that **task->parent** points to the ptracer.

Context

Must be called with **task->sighand->siglock** held.

void **task_clear_jobctl_pending**(struct task_struct * task, unsigned long mask)
clear jobctl pending bits

Parameters

struct task_struct * task target task

unsigned long mask pending bits to clear

Description

Clear **mask** from **task->jobctl**. **mask** must be subset of `JOBCTL_PENDING_MASK`. If `JOBCTL_STOP_PENDING` is being cleared, other STOP bits are cleared together.

If clearing of **mask** leaves no stop or trap pending, this function calls `task_clear_jobctl_trapping()`.

Context

Must be called with **task->sighand->siglock** held.

bool **task_participate_group_stop**(struct task_struct * task)
participate in a group stop

Parameters

struct task_struct * task task participating in a group stop

Description

task has `JOBCTL_STOP_PENDING` set and is participating in a group stop. Group stop states are cleared and the group stop count is consumed if `JOBCTL_STOP_CONSUME` was set. If the consumption completes the group stop, the appropriate SIGNAL_* flags are set.

Context

Must be called with **task->sighand->siglock** held.

Return

`true` if group stop completion should be notified to the parent, `false` otherwise.

void **ptrace_trap_notify**(struct task_struct * t)
schedule trap to notify ptracer

Parameters

struct task_struct * t tracee wanting to notify tracer

Description

This function schedules sticky ptrace trap which is cleared on the next TRAP_STOP to notify ptracer of an event. **t** must have been seized by ptracer.

If **t** is running, STOP trap will be taken. If trapped for STOP and ptracer is listening for events, tracee is woken up so that it can re-trap for the new event. If trapped otherwise, STOP trap will be eventually taken without returning to userland after the existing traps are finished by PTRACE_CONT.

Context

Must be called with **task->sighand->siglock** held.

void **do_notify_parent_cldstop**(struct task_struct * tsk, bool for_ptracer, int why)
notify parent of stopped/continued state change

Parameters

struct task_struct * tsk task reporting the state change

bool for_ptracer the notification is for ptracer

int why CLD_{CONTINUED|STOPPED|TRAPPED} to report

Description

Notify **tsk**'s parent that the stopped/continued state has changed. If **for_ptracer** is `false`, **tsk**'s group leader notifies to its real parent. If `true`, **tsk** reports to **tsk->parent** which should be the ptracer.

Context

Must be called with tasklist_lock at least read locked.

bool **do_signal_stop**(int signr)
handle group stop for SIGSTOP and other stop signals

Parameters

int signr signr causing group stop if initiating

Description

If `JOBCTL_STOP_PENDING` is not set yet, initiate group stop with **signr** and participate in it. If already set, participate in the existing group stop. If participated in a group stop (and thus slept), `true` is returned with siglock released.

If ptraced, this function doesn't handle stop itself. Instead, `JOBCTL_TRAP_STOP` is scheduled and `false` is returned with siglock untouched. The caller must ensure that INTERRUPT trap handling takes places afterwards.

Context

Must be called with **current->sighand->siglock** held, which is released on `true` return.

Return

`false` if group stop is already cancelled or ptrace trap is scheduled. `true` if participated in group stop.

void **do_jobctl_trap**(void)
take care of ptrace jobctl traps

Parameters

void no arguments

Description

When PT_SEIZED, it's used for both group stop and explicit SEIZE/INTERRUPT traps. Both generate PTRACE_EVENT_STOP trap with accompanying siginfo. If stopped, lower eight bits of exit_code contain the stop signal; otherwise, `SIGTRAP`.

When !PT_SEIZED, it's used only for group stop trap with stop signal number as exit_code and no siginfo.

Context

Must be called with **current->sighand->siglock** held, which may be released and re-acquired before returning with intervening sleep.

void **do_freezer_trap**(void)
handle the freezer jobctl trap

Parameters

void no arguments

Description

Puts the task into frozen state, if only the task is not about to quit. In this case it drops JOBCTL_TRAP_FREEZE.

Context

Must be called with **current->sighand->siglock** held, which is always released before returning.

void **signal_delivered**(struct ksignal * ksig, int stepping)

Parameters

struct ksignal * ksig kernel signal struct

int stepping nonzero if debugger single-step or block-step in use

Description

This function should be called when a signal has successfully been delivered. It updates the blocked signals accordingly (**ksig->ka.sa.sa_mask** is always blocked, and the signal itself is blocked unless `SA_NODEFER` is set in **ksig->ka.sa.sa_flags**. Tracing is notified.

long **sys_restart_syscall**(void)
restart a system call

Parameters

void no arguments

void **set_current_blocked**(sigset_t * newset)
change current->blocked mask

Parameters

sigset_t * newset new mask

Description

It is wrong to change ->blocked directly, this helper should be used to ensure the process can't miss a shared signal we are going to block.

long **sys_rt_sigprocmask**(int how, sigset_t __user * nset, sigset_t __user * oset, size_t sigsetsize)
change the list of currently blocked signals

Parameters

int how whether to add, remove, or set signals

sigset_t __user * nset stores pending signals

sigset_t __user * oset previous value of signal mask if non-null

size_t sigsetsize size of sigset_t type

long **sys_rt_sigpending**(sigset_t __user * uset, size_t sigsetsize)
examine a pending signal that has been raised while blocked

Parameters

sigset_t __user * uset stores pending signals

size_t sigsetsize size of sigset_t type or larger

void **copy_siginfo_to_external32**(struct compat_siginfo * to, const struct kernel_siginfo * from)
copy a kernel siginfo into a compat user siginfo

Parameters

struct compat_siginfo * to compat siginfo destination

const struct kernel_siginfo * from kernel siginfo source

Note

This function does not work properly for the SIGCHLD on x32, but fortunately it doesn't have to. The only valid callers for this function are copy_siginfo_to_user32, which is overriden for x32 and the coredump code. The latter does not care because SIGCHLD will never cause a coredump.

int **do_sigtimedwait**(const sigset_t * which, kernel_siginfo_t * info, const struct timespec64 * ts)
wait for queued signals specified in **which**

Parameters

const sigset_t * which queued signals to wait for

kernel_siginfo_t * info if non-null, the signal's siginfo is returned here

const struct timespec64 * ts upper bound on process time suspension

long **sys_rt_sigtimedwait**(const sigset_t __user * uthese, siginfo_t __user * uinfo, const struct __kernel_timespec __user * uts, size_t sigsetsize)
synchronously wait for queued signals specified in **uthese**

Parameters

const sigset_t __user * uthese queued signals to wait for

siginfo_t __user * uinfo if non-null, the signal's siginfo is returned here

const struct __kernel_timespec __user * uts upper bound on process time suspension

size_t sigsetsize size of sigset_t type

long **sys_kill**(pid_t pid, int sig)
send a signal to a process

Parameters

pid_t pid the PID of the process

int sig signal to be sent

long **sys_pidfd_send_signal**(int pidfd, int sig, siginfo_t __user * info, unsigned int flags)
Signal a process through a pidfd

Parameters

int pidfd file descriptor of the process

int sig signal to send

siginfo_t __user * info signal info

unsigned int flags future flags

Description

The syscall currently only signals via PIDTYPE_PID which covers kill(<positive-pid>, <signal>. It does not signal threads or process groups. In order to extend the syscall to threads and process groups the **flags** argument should be used. In essence, the **flags** argument will determine what is signaled and not the file descriptor itself. Put in other words, grouping is a property of the flags argument not a property of the file descriptor.

Return

0 on success, negative errno on failure

long **sys_tgkill**(pid_t tgid, pid_t pid, int sig)
send signal to one specific thread

Parameters

pid_t tgid the thread group ID of the thread

pid_t pid the PID of the thread

int sig signal to be sent

This syscall also checks the **tgid** and returns -ESRCH even if the PID exists but it's not belonging to the target process anymore. This method solves the problem of threads exiting and PIDs getting reused.

long **sys_tkill**(pid_t pid, int sig)
send signal to one specific task

Parameters

pid_t pid the PID of the task

int sig signal to be sent

Send a signal to only one task, even if it's a CLONE_THREAD task.

long **sys_rt_sigqueueinfo**(pid_t pid, int sig, siginfo_t __user * uinfo)
send signal information to a signal

Parameters

pid_t pid the PID of the thread

int sig signal to be sent

siginfo_t __user * uinfo signal info to be sent

long **sys_sigpending**(old_sigset_t __user * uset)
examine pending signals

Parameters

old_sigset_t __user * uset where mask of pending signal is returned

long **sys_sigprocmask**(int how, old_sigset_t __user * nset, old_sigset_t __user * oset)
examine and change blocked signals

Parameters

int how whether to add, remove, or set signals

old_sigset_t __user * nset signals to add or remove (if non-null)

old_sigset_t __user * oset previous value of signal mask if non-null

Description

Some platforms have their own version with special arguments; others support only sys_rt_sigprocmask.

long **sys_rt_sigaction**(int sig, const struct sigaction __user * act, struct sigaction __user * oact, size_t sigsetsize)
alter an action taken by a process

Parameters

int sig signal to be sent

const struct sigaction __user * act new sigaction

struct sigaction __user * oact used to save the previous sigaction

size_t sigsetsize size of sigset_t type

long **sys_rt_sigsuspend**(sigset_t __user * unewset, size_t sigsetsize)
replace the signal mask for a value with the **unewset** value until a signal is received

Parameters

sigset_t __user * unewset new signal mask value

size_t sigsetsize size of sigset_t type

kthread_create(threadfn, data, namefmt, arg)
create a kthread on the current node

Parameters

threadfn the function to run in the thread

data data pointer for **threadfn()**

namefmt printf-style format string for the thread name

arg arguments for **namefmt**.

Description

This macro will create a kthread on the current node, leaving it in the stopped state. This is just a helper for `kthread_create_on_node()`; see the documentation there for more details.

kthread_run(threadfn, data, namefmt, ⋯)
create and wake a thread.

Parameters

threadfn the function to run until signal_pending(current).

data data ptr for **threadfn**.

namefmt printf-style name for the thread.

... variable arguments

Description

Convenient wrapper for `kthread_create()` followed by `wake_up_process()`. Returns the kthread or ERR_PTR(-ENOMEM).

bool **kthread_should_stop**(void)
should this kthread return now?

Parameters

void no arguments

Description

When someone calls `kthread_stop()` on your kthread, it will be woken and this will return true. You should then return, and your return value will be passed through to `kthread_stop()`.

bool **kthread_should_park**(void)
should this kthread park now?

Parameters

void no arguments

Description

When someone calls `kthread_park()` on your kthread, it will be woken and this will return true. You should then do the necessary cleanup and call kthread_parkme()

Similar to `kthread_should_stop()`, but this keeps the thread alive and in a park position. `kthread_unpark()` "restarts" the thread and calls the thread function again.

bool **kthread_freezable_should_stop**(bool * was_frozen)
should this freezable kthread return now?

Parameters

bool * was_frozen optional out parameter, indicates whether `current` was frozen

Description

`kthread_should_stop()` for freezable kthreads, which will enter refrigerator if necessary. This function is safe from `kthread_stop()` / freezer deadlock and freezable kthreads should use this function instead of calling try_to_freeze() directly.

void * **kthread_func**(struct task_struct * task)
return the function specified on kthread creation

Parameters

struct task_struct * task kthread task in question

Description

Returns NULL if the task is not a kthread.

void * **kthread_data**(struct task_struct * task)
return data value specified on kthread creation

Parameters

struct task_struct * task kthread task in question

Description

Return the data value specified when kthread **task** was created. The caller is responsible for ensuring the validity of **task** when calling this function.

struct task_struct * **kthread_create_on_node**(int (*threadfn)(void *data), void * data, int node, const char namefmt, ...)
create a kthread.

Parameters

int (*)(void *data) threadfn the function to run until signal_pending(current).

void * data data ptr for **threadfn**.

int node task and thread structures for the thread are allocated on this node

const char namefmt printf-style name for the thread.

... variable arguments

Description

This helper function creates and names a kernel thread. The thread will be stopped: use wake_up_process() to start it. See also kthread_run(). The new thread has SCHED_NORMAL policy and is affine to all CPUs.

If thread is going to be bound on a particular cpu, give its node in **node**, to get NUMA affinity for kthread stack, or else give NUMA_NO_NODE. When woken, the thread will run **threadfn()** with **data** as its argument. **threadfn()** can either call do_exit() directly if it is a standalone thread for which no one will call kthread_stop(), or return when 'kthread_should_stop()' is true (which means kthread_stop() has been called). The return value should be zero or a negative error number; it will be passed to kthread_stop().

Returns a task_struct or ERR_PTR(-ENOMEM) or ERR_PTR(-EINTR).

void **kthread_bind**(struct task_struct * p, unsigned int cpu)
bind a just-created kthread to a cpu.

Parameters

struct task_struct * p thread created by kthread_create().

unsigned int cpu cpu (might not be online, must be possible) for **k** to run on.

Description

This function is equivalent to set_cpus_allowed(), except that **cpu** doesn't need to be online, and the thread must be stopped (i.e., just returned from kthread_create()).

void **kthread_unpark**(struct task_struct * k)
unpark a thread created by kthread_create().

Parameters

struct task_struct * k thread created by kthread_create().

Description

Sets kthread_should_park() for **k** to return false, wakes it, and waits for it to return. If the thread is marked percpu then its bound to the cpu again.

int **kthread_park**(struct task_struct * k)
park a thread created by kthread_create().

Parameters

struct task_struct * k thread created by kthread_create().

Description

Sets kthread_should_park() for **k** to return true, wakes it, and waits for it to return. This can also be called after kthread_create() instead of calling wake_up_process(): the thread will park without calling threadfn().

Returns 0 if the thread is parked, -ENOSYS if the thread exited. If called by the kthread itself just the park bit is set.

int **kthread_stop**(struct task_struct * k)
stop a thread created by `kthread_create()`.

Parameters

struct task_struct * k thread created by `kthread_create()`.

Description

Sets `kthread_should_stop()` for **k** to return true, wakes it, and waits for it to exit. This can also be called after `kthread_create()` instead of calling `wake_up_process()`: the thread will exit without calling threadfn().

If threadfn() may call do_exit() itself, the caller must ensure task_struct can't go away.

Returns the result of threadfn(), or `-EINTR` if `wake_up_process()` was never called.

int **kthread_worker_fn**(void * worker_ptr)
kthread function to process kthread_worker

Parameters

void * worker_ptr pointer to initialized kthread_worker

Description

This function implements the main cycle of kthread worker. It processes work_list until it is stopped with `kthread_stop()`. It sleeps when the queue is empty.

The works are not allowed to keep any locks, disable preemption or interrupts when they finish. There is defined a safe point for freezing when one work finishes and before a new one is started.

Also the works must not be handled by more than one worker at the same time, see also `kthread_queue_work()`.

struct kthread_worker * **kthread_create_worker**(unsigned int flags, const char namefmt, ...)
create a kthread worker

Parameters

unsigned int flags flags modifying the default behavior of the worker

const char namefmt printf-style name for the kthread worker (task).

... variable arguments

Description

Returns a pointer to the allocated worker on success, ERR_PTR(-ENOMEM) when the needed structures could not get allocated, and ERR_PTR(-EINTR) when the worker was SIGKILLed.

struct kthread_worker * **kthread_create_worker_on_cpu**(int cpu, unsigned int flags, const char namefmt, ...)
create a kthread worker and bind it it to a given CPU and the associated NUMA node.

Parameters

int cpu CPU number

unsigned int flags flags modifying the default behavior of the worker

const char namefmt printf-style name for the kthread worker (task).

... variable arguments

Description

Use a valid CPU number if you want to bind the kthread worker to the given CPU and the associated NUMA node.

A good practice is to add the cpu number also into the worker name. For example, use kthread_create_worker_on_cpu(cpu, "helper/d", cpu).

Returns a pointer to the allocated worker on success, ERR_PTR(-ENOMEM) when the needed structures could not get allocated, and ERR_PTR(-EINTR) when the worker was SIGKILLed.

bool **kthread_queue_work**(struct kthread_worker * worker, struct kthread_work * work)
queue a kthread_work

Parameters

struct kthread_worker * worker target kthread_worker

struct kthread_work * work kthread_work to queue

Description

Queue **work** to work processor **task** for async execution. **task** must have been created with kthread_worker_create(). Returns `true` if **work** was successfully queued, `false` if it was already pending.

Reinitialize the work if it needs to be used by another worker. For example, when the worker was stopped and started again.

void **kthread_delayed_work_timer_fn**(struct timer_list * t)
callback that queues the associated kthread delayed work when the timer expires.

Parameters

struct timer_list * t pointer to the expired timer

Description

The format of the function is defined by struct timer_list. It should have been called from irqsafe timer with irq already off.

bool **kthread_queue_delayed_work**(struct kthread_worker * worker, struct kthread_delayed_work * dwork, unsigned long delay)
queue the associated kthread work after a delay.

Parameters

struct kthread_worker * worker target kthread_worker

struct kthread_delayed_work * dwork kthread_delayed_work to queue

unsigned long delay number of jiffies to wait before queuing

Description

If the work has not been pending it starts a timer that will queue the work after the given **delay**. If **delay** is zero, it queues the work immediately.

Return

`false` if the **work** has already been pending. It means that either the timer was running or the work was queued. It returns `true` otherwise.

void **kthread_flush_work**(struct kthread_work * work)
flush a kthread_work

Parameters

struct kthread_work * work work to flush

Description

If **work** is queued or executing, wait for it to finish execution.

bool **kthread_mod_delayed_work**(struct kthread_worker * worker, struct kthread_delayed_work * dwork, unsigned long delay)
modify delay of or queue a kthread delayed work

Parameters

struct kthread_worker * worker kthread worker to use

struct kthread_delayed_work * dwork kthread delayed work to queue

unsigned long delay number of jiffies to wait before queuing

Description

If **dwork** is idle, equivalent to `kthread_queue_delayed_work()`. Otherwise, modify **dwork**'s timer so that it expires after **delay**. If **delay** is zero, **work** is guaranteed to be queued immediately.

A special case is when the work is being canceled in parallel. It might be caused either by the real `kthread_cancel_delayed_work_sync()` or yet another `kthread_mod_delayed_work()` call. We let the other command win and return `false` here. The caller is supposed to synchronize these operations a reasonable way.

This function is safe to call from any context including IRQ handler. See __kthread_cancel_work() and `kthread_delayed_work_timer_fn()` for details.

Return

`true` if **dwork** was pending and its timer was modified, `false` otherwise.

bool **kthread_cancel_work_sync**(struct kthread_work * work)
cancel a kthread work and wait for it to finish

Parameters

struct kthread_work * work the kthread work to cancel

Description

Cancel **work** and wait for its execution to finish. This function can be used even if the work re-queues itself. On return from this function, **work** is guaranteed to be not pending or executing on any CPU.

kthread_cancel_work_sync(`delayed_work->work`) must not be used for delayed_work's. Use `kthread_cancel_delayed_work_sync()` instead.

The caller must ensure that the worker on which **work** was last queued can't be destroyed before this function returns.

Return

`true` if **work** was pending, `false` otherwise.

bool **kthread_cancel_delayed_work_sync**(struct kthread_delayed_work * dwork)
cancel a kthread delayed work and wait for it to finish.

Parameters

struct kthread_delayed_work * dwork the kthread delayed work to cancel

Description

This is `kthread_cancel_work_sync()` for delayed works.

Return

`true` if **dwork** was pending, `false` otherwise.

void **kthread_flush_worker**(struct kthread_worker * worker)
flush all current works on a kthread_worker

Parameters

struct kthread_worker * worker worker to flush

Description

Wait until all currently executing or pending works on **worker** are finished.

void **kthread_destroy_worker**(struct kthread_worker * worker)
destroy a kthread worker

Parameters

struct kthread_worker * worker worker to be destroyed

Description

Flush and destroy **worker**. The simple flush is enough because the kthread worker API is used only in trivial scenarios. There are no multi-step state machines needed.

void **kthread_use_mm**(struct mm_struct * mm)
make the calling kthread operate on an address space

Parameters

struct mm_struct * mm address space to operate on

void **kthread_unuse_mm**(struct mm_struct * mm)
reverse the effect of `kthread_use_mm()`

Parameters

struct mm_struct * mm address space to operate on

void **kthread_associate_blkcg**(struct cgroup_subsys_state * css)
associate blkcg to current kthread

Parameters

struct cgroup_subsys_state * css the cgroup info

Description

Current thread must be a kthread. The thread is running jobs on behalf of other threads. In some cases, we expect the jobs attach cgroup info of original threads instead of that of current thread. This function stores original thread's cgroup info in current kthread context for later retrieval.

struct cgroup_subsys_state * **kthread_blkcg**(void)
get associated blkcg css of current kthread

Parameters

void no arguments

Description

Current thread must be a kthread.

2.8 Reference counting

struct **refcount_struct**
variant of atomic_t specialized for reference counts

Definition

```
struct refcount_struct {
  atomic_t refs;
};
```

Members

refs atomic_t counter field

Description

The counter saturates at REFCOUNT_SATURATED and will not move once there. This avoids wrapping the counter and causing 'spurious' use-after-free bugs.

void **refcount_set**(refcount_t * r, int n)
set a refcount's value

Parameters

refcount_t * r the refcount

int n value to which the refcount will be set

unsigned int **refcount_read**(const refcount_t * r)
get a refcount’s value

Parameters

const refcount_t * r the refcount

Return

the refcount’s value

bool **refcount_add_not_zero**(int i, refcount_t * r)
add a value to a refcount unless it is 0

Parameters

int i the value to add to the refcount

refcount_t * r the refcount

Description

Will saturate at REFCOUNT_SATURATED and WARN.

Provides no memory ordering, it is assumed the caller has guaranteed the object memory to be stable (RCU, etc.). It does provide a control dependency and thereby orders future stores. See the comment on top.

Use of this function is not recommended for the normal reference counting use case in which references are taken and released one at a time. In these cases, `refcount_inc()`, or one of its variants, should instead be used to increment a reference count.

Return

false if the passed refcount is 0, true otherwise

void **refcount_add**(int i, refcount_t * r)
add a value to a refcount

Parameters

int i the value to add to the refcount

refcount_t * r the refcount

Description

Similar to atomic_add(), but will saturate at REFCOUNT_SATURATED and WARN.

Provides no memory ordering, it is assumed the caller has guaranteed the object memory to be stable (RCU, etc.). It does provide a control dependency and thereby orders future stores. See the comment on top.

Use of this function is not recommended for the normal reference counting use case in which references are taken and released one at a time. In these cases, `refcount_inc()`, or one of its variants, should instead be used to increment a reference count.

bool **refcount_inc_not_zero**(refcount_t * r)
increment a refcount unless it is 0

Parameters

refcount_t * r the refcount to increment

Description

Similar to atomic_inc_not_zero(), but will saturate at REFCOUNT_SATURATED and WARN.

Provides no memory ordering, it is assumed the caller has guaranteed the object memory to be stable (RCU, etc.). It does provide a control dependency and thereby orders future stores. See the comment on top.

Return

true if the increment was successful, false otherwise

void **refcount_inc**(refcount_t * r)
increment a refcount

Parameters

refcount_t * r the refcount to increment

Description

Similar to atomic_inc(), but will saturate at REFCOUNT_SATURATED and WARN.

Provides no memory ordering, it is assumed the caller already has a reference on the object.

Will WARN if the refcount is 0, as this represents a possible use-after-free condition.

bool **refcount_sub_and_test**(int i, refcount_t * r)
subtract from a refcount and test if it is 0

Parameters

int i amount to subtract from the refcount

refcount_t * r the refcount

Description

Similar to atomic_dec_and_test(), but it will WARN, return false and ultimately leak on underflow and will fail to decrement when saturated at REFCOUNT_SATURATED.

Provides release memory ordering, such that prior loads and stores are done before, and provides an acquire ordering on success such that free() must come after.

Use of this function is not recommended for the normal reference counting use case in which references are taken and released one at a time. In these cases, `refcount_dec()`, or one of its variants, should instead be used to decrement a reference count.

Return

true if the resulting refcount is 0, false otherwise

bool **refcount_dec_and_test**(refcount_t * r)
decrement a refcount and test if it is 0

Parameters

refcount_t * r the refcount

Description

Similar to atomic_dec_and_test(), it will WARN on underflow and fail to decrement when saturated at REFCOUNT_SATURATED.

Provides release memory ordering, such that prior loads and stores are done before, and provides an acquire ordering on success such that free() must come after.

Return

true if the resulting refcount is 0, false otherwise

void **refcount_dec**(refcount_t * r)
decrement a refcount

Parameters

refcount_t * r the refcount

Description

Similar to atomic_dec(), it will WARN on underflow and fail to decrement when saturated at REFCOUNT_SATURATED.

Provides release memory ordering, such that prior loads and stores are done before.

bool **refcount_dec_if_one**(refcount_t * r)
decrement a refcount if it is 1

Parameters

refcount_t * r the refcount

Description

No atomic_t counterpart, it attempts a 1 -> 0 transition and returns the success thereof.

Like all decrement operations, it provides release memory order and provides a control dependency.

It can be used like a try-delete operator; this explicit case is provided and not cmpxchg in generic, because that would allow implementing unsafe operations.

Return

true if the resulting refcount is 0, false otherwise

bool **refcount_dec_not_one**(refcount_t * r)
decrement a refcount if it is not 1

Parameters

refcount_t * r the refcount

Description

No atomic_t counterpart, it decrements unless the value is 1, in which case it will return false.

Was often done like: atomic_add_unless(var, -1, 1)

Return

true if the decrement operation was successful, false otherwise

bool **refcount_dec_and_mutex_lock**(refcount_t * r, struct mutex * lock)
return holding mutex if able to decrement refcount to 0

Parameters

refcount_t * r the refcount

struct mutex * lock the mutex to be locked

Description

Similar to atomic_dec_and_mutex_lock(), it will WARN on underflow and fail to decrement when saturated at REFCOUNT_SATURATED.

Provides release memory ordering, such that prior loads and stores are done before, and provides a control dependency such that free() must come after. See the comment on top.

Return

true and hold mutex if able to decrement refcount to 0, false otherwise

bool **refcount_dec_and_lock**(refcount_t * r, spinlock_t * lock)
return holding spinlock if able to decrement refcount to 0

Parameters

refcount_t * r the refcount

spinlock_t * lock the spinlock to be locked

Description

Similar to atomic_dec_and_lock(), it will WARN on underflow and fail to decrement when saturated at REFCOUNT_SATURATED.

Provides release memory ordering, such that prior loads and stores are done before, and provides a control dependency such that free() must come after. See the comment on top.

Return

true and hold spinlock if able to decrement refcount to 0, false otherwise

bool **refcount_dec_and_lock_irqsave**(refcount_t * r, spinlock_t * lock, unsigned long * flags)
return holding spinlock with disabled interrupts if able to decrement refcount to 0

Parameters

refcount_t * r the refcount

spinlock_t * lock the spinlock to be locked

unsigned long * flags saved IRQ-flags if the is acquired

Description

Same as `refcount_dec_and_lock()` above except that the spinlock is acquired with disabled interupts.

Return

true and hold spinlock if able to decrement refcount to 0, false otherwise

2.9 Atomics

int **arch_atomic_read**(const atomic_t * v)
read atomic variable

Parameters

const atomic_t * v pointer of type atomic_t

Description

Atomically reads the value of **v**.

void **arch_atomic_set**(atomic_t * v, int i)
set atomic variable

Parameters

atomic_t * v pointer of type atomic_t

int i required value

Description

Atomically sets the value of **v** to **i**.

void **arch_atomic_add**(int i, atomic_t * v)
add integer to atomic variable

Parameters

int i integer value to add

atomic_t * v pointer of type atomic_t

Description

Atomically adds **i** to **v**.

void **arch_atomic_sub**(int i, atomic_t * v)
subtract integer from atomic variable

Parameters

int i integer value to subtract

atomic_t * v pointer of type atomic_t

Description

Atomically subtracts **i** from **v**.

bool **arch_atomic_sub_and_test**(int i, atomic_t * v)
subtract value from variable and test result

Parameters

int i integer value to subtract

atomic_t * v pointer of type atomic_t

Description

Atomically subtracts **i** from **v** and returns true if the result is zero, or false for all other cases.

void **arch_atomic_inc**(atomic_t * v)
increment atomic variable

Parameters

atomic_t * v pointer of type atomic_t

Description

Atomically increments **v** by 1.

void **arch_atomic_dec**(atomic_t * v)
decrement atomic variable

Parameters

atomic_t * v pointer of type atomic_t

Description

Atomically decrements **v** by 1.

bool **arch_atomic_dec_and_test**(atomic_t * v)
decrement and test

Parameters

atomic_t * v pointer of type atomic_t

Description

Atomically decrements **v** by 1 and returns true if the result is 0, or false for all other cases.

bool **arch_atomic_inc_and_test**(atomic_t * v)
increment and test

Parameters

atomic_t * v pointer of type atomic_t

Description

Atomically increments **v** by 1 and returns true if the result is zero, or false for all other cases.

bool **arch_atomic_add_negative**(int i, atomic_t * v)
add and test if negative

Parameters

int i integer value to add

atomic_t * v pointer of type atomic_t

Description

Atomically adds **i** to **v** and returns true if the result is negative, or false when result is greater than or equal to zero.

int **arch_atomic_add_return**(int i, atomic_t * v)
add integer and return

Parameters

int i integer value to add

atomic_t * v pointer of type atomic_t

Description

Atomically adds **i** to **v** and returns **i** + **v**

int **arch_atomic_sub_return**(int i, atomic_t * v)
subtract integer and return

Parameters

int i integer value to subtract

atomic_t * v pointer of type atomic_t

Description

Atomically subtracts **i** from **v** and returns **v** - **i**

2.10 Kernel objects manipulation

char * **kobject_get_path**(struct kobject * kobj, gfp_t gfp_mask)
Allocate memory and fill in the path for **kobj**.

Parameters

struct kobject * kobj kobject in question, with which to build the path

gfp_t gfp_mask the allocation type used to allocate the path

Return

The newly allocated memory, caller must free with kfree().

int **kobject_set_name**(struct kobject * kobj, const char * fmt, ...)
Set the name of a kobject.

Parameters

struct kobject * kobj struct kobject to set the name of

const char * fmt format string used to build the name

... variable arguments

Description

This sets the name of the kobject. If you have already added the kobject to the system, you must call `kobject_rename()` in order to change the name of the kobject.

void **kobject_init**(struct kobject * kobj, struct kobj_type * ktype)
Initialize a kobject structure.

Parameters

struct kobject * kobj pointer to the kobject to initialize

struct kobj_type * ktype pointer to the ktype for this kobject.

Description

This function will properly initialize a kobject such that it can then be passed to the `kobject_add()` call.

After this function is called, the kobject MUST be cleaned up by a call to `kobject_put()`, not by a call to kfree directly to ensure that all of the memory is cleaned up properly.

int **kobject_add**(struct kobject * kobj, struct kobject * parent, const char * fmt, ...)
The main kobject add function.

Parameters

struct kobject * kobj the kobject to add

struct kobject * parent pointer to the parent of the kobject.

const char * fmt format to name the kobject with.

... variable arguments

Description

The kobject name is set and added to the kobject hierarchy in this function.

If **parent** is set, then the parent of the **kobj** will be set to it. If **parent** is NULL, then the parent of the **kobj** will be set to the kobject associated with the kset assigned to this kobject. If no kset is assigned to the kobject, then the kobject will be located in the root of the sysfs tree.

Note, no "add" uevent will be created with this call, the caller should set up all of the necessary sysfs files for the object and then call kobject_uevent() with the UEVENT_ADD parameter to ensure that userspace is properly notified of this kobject's creation.

Return

If this function returns an error, kobject_put() must be called to properly clean up the memory associated with the object. Under no instance should the kobject that is passed to this function be directly freed with a call to kfree(), that can leak memory.

If this function returns success, `kobject_put()` must also be called in order to properly clean up the memory associated with the object.

In short, once this function is called, `kobject_put()` MUST be called when the use of the object is finished in order to properly free everything.

int **kobject_init_and_add**(struct kobject * kobj, struct kobj_type * ktype, struct kobject * parent, const char * fmt, ...)
Initialize a kobject structure and add it to the kobject hierarchy.

Parameters

struct kobject * kobj pointer to the kobject to initialize

struct kobj_type * ktype pointer to the ktype for this kobject.

struct kobject * parent pointer to the parent of this kobject.

const char * fmt the name of the kobject.

... variable arguments

Description

This function combines the call to `kobject_init()` and `kobject_add()`.

If this function returns an error, `kobject_put()` must be called to properly clean up the memory associated with the object. This is the same type of error handling after a call to `kobject_add()` and kobject lifetime rules are the same here.

int **kobject_rename**(struct kobject * kobj, const char * new_name)
Change the name of an object.

Parameters

struct kobject * kobj object in question.

const char * new_name object' s new name

Description

It is the responsibility of the caller to provide mutual exclusion between two different calls of kobject_rename on the same kobject and to ensure that new_name is valid and won' t conflict with other kobjects.

int **kobject_move**(struct kobject * kobj, struct kobject * new_parent)
Move object to another parent.

Parameters

struct kobject * kobj object in question.

struct kobject * new_parent object' s new parent (can be NULL)

void **kobject_del**(struct kobject * kobj)
Unlink kobject from hierarchy.

Parameters

struct kobject * kobj object.

Description

This is the function that should be called to delete an object successfully added via `kobject_add()`.

struct kobject * **kobject_get**(struct kobject * kobj)
Increment refcount for object.

Parameters

struct kobject * kobj object.

void **kobject_put**(struct kobject * kobj)
Decrement refcount for object.

Parameters

struct kobject * kobj object.

Description

Decrement the refcount, and if 0, call kobject_cleanup().

struct kobject * **kobject_create_and_add**(const char * name, struct kobject * parent)
Create a struct kobject dynamically and register it with sysfs.

Parameters

const char * name the name for the kobject

struct kobject * parent the parent kobject of this kobject, if any.

Description

This function creates a kobject structure dynamically and registers it with sysfs. When you are finished with this structure, call `kobject_put()` and the structure will be dynamically freed when it is no longer being used.

If the kobject was not able to be created, NULL will be returned.

int **kset_register**(struct kset * k)
Initialize and add a kset.

Parameters

struct kset * k kset.

void **kset_unregister**(struct kset * k)
Remove a kset.

Parameters

struct kset * k kset.

struct kobject * **kset_find_obj**(struct kset * kset, const char * name)
Search for object in kset.

Parameters

struct kset * kset kset we' re looking in.

const char * name object' s name.

Description

Lock kset via **kset->subsys**, and iterate over **kset->list**, looking for a matching kobject. If matching object is found take a reference and return the object.

struct kset * **kset_create_and_add**(const char * name, const struct kset_uevent_ops * uevent_ops, struct kobject * parent_kobj)
Create a struct kset dynamically and add it to sysfs.

Parameters

const char * name the name for the kset

const struct kset_uevent_ops * uevent_ops a struct kset_uevent_ops for the kset

struct kobject * parent_kobj the parent kobject of this kset, if any.

Description

This function creates a kset structure dynamically and registers it with sysfs. When you are finished with this structure, call `kset_unregister()` and the structure will be dynamically freed when it is no longer being used.

If the kset was not able to be created, NULL will be returned.

2.11 Kernel utility functions

REPEAT_BYTE(x)
repeat the value **x** multiple times as an unsigned long value

Parameters

x value to repeat

NOTE

x is not checked for > 0xff; larger values produce odd results.

ARRAY_SIZE(arr)
get the number of elements in array **arr**

Parameters

arr array to be sized

round_up(x, y)
round up to next specified power of 2

Parameters

x the value to round

y multiple to round up to (must be a power of 2)

Description

Rounds **x** up to next multiple of **y** (which must be a power of 2). To perform arbitrary rounding up, use `roundup()` below.

round_down(x, y)
round down to next specified power of 2

Parameters

x the value to round

y multiple to round down to (must be a power of 2)

Description

Rounds **x** down to next multiple of **y** (which must be a power of 2). To perform arbitrary rounding down, use `rounddown()` below.

roundup(x, y)
round up to the next specified multiple

Parameters

x the value to up

y multiple to round up to

Description

Rounds **x** up to next multiple of **y**. If **y** will always be a power of 2, consider using the faster `round_up()`.

rounddown(x, y)
round down to next specified multiple

Parameters

x the value to round

y multiple to round down to

Description

Rounds **x** down to next multiple of **y**. If **y** will always be a power of 2, consider using the faster `round_down()`.

upper_32_bits(n)
return bits 32-63 of a number

Parameters

n the number we' re accessing

Description

A basic shift-right of a 64- or 32-bit quantity. Use this to suppress the "right shift count >= width of type" warning when that quantity is 32-bits.

lower_32_bits(n)
return bits 0-31 of a number

Parameters

n the number we' re accessing

might_sleep()
annotation for functions that can sleep

Parameters

Description

this macro will print a stack trace if it is executed in an atomic context (spinlock, irq-handler, ⋯). Additional sections where blocking is not allowed can be annotated with `non_block_start()` and `non_block_end()` pairs.

This is a useful debugging help to be able to catch problems early and not be bitten later when the calling function happens to sleep when it is not supposed to.

cant_sleep()
annotation for functions that cannot sleep

Parameters

Description

this macro will print a stack trace if it is executed with preemption enabled

non_block_start()
annotate the start of section where sleeping is prohibited

Parameters

Description

This is on behalf of the oom reaper, specifically when it is calling the mmu notifiers. The problem is that if the notifier were to block on, for example, mutex_lock() and if the process which holds that mutex were to perform a sleeping memory allocation, the oom reaper is now blocked on completion of that memory allocation. Other blocking calls like `wait_event()` pose similar issues.

non_block_end()
annotate the end of section where sleeping is prohibited

Parameters

Description

Closes a section opened by `non_block_start()`.

abs(x)
return absolute value of an argument

Parameters

x the value. If it is unsigned type, it is converted to signed type first. char is treated as if it was signed (regardless of whether it really is) but the macro's return type is preserved as char.

Return

an absolute value of x.

u32 **reciprocal_scale**(u32 val, u32 ep_ro)
"scale" a value into range [0, ep_ro)

Parameters

u32 val value

u32 ep_ro right open interval endpoint

Description

Perform a "reciprocal multiplication" in order to "scale" a value into range [0, **ep_ro**), where the upper interval endpoint is right-open. This is useful, e.g. for accessing a index of an array containing **ep_ro** elements, for example. Think of it as sort of modulus, only that the result isn't that of modulo. ;) Note that if initial input is a small value, then result will return 0.

Return

a result based on **val** in interval [0, **ep_ro**).

int **kstrtoul**(const char * s, unsigned int base, unsigned long * res)
convert a string to an unsigned long

Parameters

const char * s The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign, but not a minus sign.

unsigned int base The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

unsigned long * res Where to write the result of the conversion on success.

Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Used as a replacement for the simple_strtoull. Return code must be checked.

int **kstrtol**(const char * s, unsigned int base, long * res)
convert a string to a long

Parameters

const char * s The start of the string. The string must be null-terminated, and may also include a single newline before its terminating null. The first character may also be a plus sign or a minus sign.

unsigned int base The number base to use. The maximum supported base is 16. If base is given as 0, then the base of the string is automatically detected with the conventional semantics - If it begins with 0x the number will be parsed as a hexadecimal (case insensitive), if it otherwise begins with 0, it will be parsed as an octal number. Otherwise it will be parsed as a decimal.

long * res Where to write the result of the conversion on success.

Description

Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error. Used as a replacement for the simple_strtoull. Return code must be checked.

trace_printk(fmt, ⋯)
printf formatting in the ftrace buffer

Parameters

fmt the printf format for printing

... variable arguments

Note

__trace_printk is an internal function for trace_printk() and the **ip** is passed in via the `trace_printk()` macro.

Description

This function allows a kernel developer to debug fast path sections that printk is not appropriate for. By scattering in various printk like tracing in the code, a developer can quickly see where problems are occurring.

This is intended as a debugging tool for the developer only. Please refrain from leaving trace_printks scattered around in your code. (Extra memory is used for special buffers that are allocated when `trace_printk()` is used.)

A little optimization trick is done here. If there's only one argument, there's no need to scan the string for printf formats. The `trace_puts()` will suffice. But how can we take advantage of using `trace_puts()` when `trace_printk()` has only one argument? By stringifying the args and checking the size we can tell whether or not there are args. __stringify((__VA_ARGS__)) will turn into "()0" with a size of 3 when there are no args, anything else will be bigger. All we need to do is define a string to this, and then take its size and compare to 3. If it's bigger, use do_trace_printk() otherwise, optimize it to `trace_puts()`. Then just let gcc optimize the rest.

trace_puts(str)
write a string into the ftrace buffer

Parameters

str the string to record

Note

__trace_bputs is an internal function for trace_puts and the **ip** is passed in via the trace_puts macro.

Description

This is similar to `trace_printk()` but is made for those really fast paths that a developer wants the least amount of "Heisenbug" effects, where the processing of the print format is still too much.

This function allows a kernel developer to debug fast path sections that printk is not appropriate for. By scattering in various printk like tracing in the code, a developer can quickly see where problems are occurring.

This is intended as a debugging tool for the developer only. Please refrain from leaving trace_puts scattered around in your code. (Extra memory is used for special buffers that are allocated when `trace_puts()` is used.)

Return

0 if nothing was written, positive # if string was. (1 when __trace_bputs is used, strlen(str) when __trace_puts is used)

min(x, y)
return minimum of two values of the same or compatible types

Parameters

x first value

y second value

max(x, y)
return maximum of two values of the same or compatible types

Parameters

x first value

y second value

min3(x, y, z)
return minimum of three values

Parameters

x first value

y second value

z third value

max3(x, y, z)
return maximum of three values

Parameters

x first value

y second value

z third value

min_not_zero(x, y)
return the minimum that is _not_ zero, unless both are zero

Parameters

x value1

y value2

clamp(val, lo, hi)
return a value clamped to a given range with strict typechecking

Parameters

val current value

lo lowest allowable value

hi highest allowable value

Description

This macro does strict typechecking of **lo/hi** to make sure they are of the same type as **val**. See the unnecessary pointer comparisons.

min_t(type, x, y)
return minimum of two values, using the specified type

Parameters

type data type to use

x first value

y second value

max_t(type, x, y)
return maximum of two values, using the specified type

Parameters

type data type to use

x first value

y second value

clamp_t(type, val, lo, hi)
return a value clamped to a given range using a given type

Parameters

type the type of variable to use

val current value

lo minimum allowable value

hi maximum allowable value

Description

This macro does no typechecking and uses temporary variables of type **type** to make all the comparisons.

clamp_val(val, lo, hi)
return a value clamped to a given range using val' s type

Parameters

val current value

lo minimum allowable value

hi maximum allowable value

Description

This macro does no typechecking and uses temporary variables of whatever type the input argument **val** is. This is useful when **val** is an unsigned type and **lo** and **hi** are literals that will otherwise be assigned a signed integer type.

swap(a, b)
swap values of **a** and **b**

Parameters

a first value

b second value

container_of(ptr, type, member)
cast a member of a structure out to the containing structure

Parameters

ptr the pointer to the member.

type the type of the container struct this is embedded in.

member the name of the member within the struct.

container_of_safe(ptr, type, member)
cast a member of a structure out to the containing structure

Parameters

ptr the pointer to the member.

type the type of the container struct this is embedded in.

member the name of the member within the struct.

Description

If IS_ERR_OR_NULL(ptr), ptr is returned unchanged.

__visible int **printk**(const char * fmt, ...)
print a kernel message

Parameters

const char * fmt format string

... variable arguments

Description

This is `printk()`. It can be called from any context. We want it to work.

We try to grab the console_lock. If we succeed, it' s easy - we log the output and call the console drivers. If we fail to get the semaphore, we place the output into the log buffer and return. The current holder of the console_sem will notice the new output in `console_unlock()`; and will send it to the consoles before releasing the lock.

One effect of this deferred printing is that code which calls `printk()` and then changes console_loglevel may break. This is because console_loglevel is inspected when the actual printing occurs.

See also: printf(3)

See the vsnprintf() documentation for format string extensions over C99.

void **console_lock**(void)
lock the console system for exclusive use.

Parameters

void no arguments

Description

Acquires a lock which guarantees that the caller has exclusive access to the console system and the console_drivers list.

Can sleep, returns nothing.

int **console_trylock**(void)
try to lock the console system for exclusive use.

Parameters

void no arguments

Description

Try to acquire a lock which guarantees that the caller has exclusive access to the console system and the console_drivers list.

returns 1 on success, and 0 on failure to acquire the lock.

void **console_unlock**(void)
unlock the console system

Parameters

void no arguments

Description

Releases the console_lock which the caller holds on the console system and the console driver list.

While the console_lock was held, console output may have been buffered by `printk()`. If this is the case, `console_unlock();` emits the output prior to releasing the lock.

If there is output waiting, we wake /dev/kmsg and syslog() users.

`console_unlock();` may be called from any context.

void **console_conditional_schedule**(void)
yield the CPU if required

Parameters

void no arguments

Description

If the console code is currently allowed to sleep, and if this CPU should yield the CPU to another task, do so here.

Must be called within `console_lock();`.

bool **printk_timed_ratelimit**(unsigned long * caller_jiffies, unsigned int interval_msecs)
caller-controlled printk ratelimiting

Parameters

unsigned long * caller_jiffies pointer to caller's state

unsigned int interval_msecs minimum interval between prints

Description

`printk_timed_ratelimit()` returns true if more than **interval_msecs** milliseconds have elapsed since the last time `printk_timed_ratelimit()` returned true.

int **kmsg_dump_register**(struct kmsg_dumper * dumper)
register a kernel log dumper.

Parameters

struct kmsg_dumper * dumper pointer to the kmsg_dumper structure

Description

Adds a kernel log dumper to the system. The dump callback in the structure will be called when the kernel oopses or panics and must be set. Returns zero on success and `-EINVAL` or `-EBUSY` otherwise.

int **kmsg_dump_unregister**(struct kmsg_dumper * dumper)
unregister a kmsg dumper.

Parameters

struct kmsg_dumper * dumper pointer to the kmsg_dumper structure

Description

Removes a dump device from the system. Returns zero on success and -EINVAL otherwise.

bool **kmsg_dump_get_line**(struct kmsg_dumper * dumper, bool syslog, char * line, size_t size, size_t * len)
retrieve one kmsg log line

Parameters

struct kmsg_dumper * dumper registered kmsg dumper

bool syslog include the “<4>” prefixes

char * line buffer to copy the line to

size_t size maximum size of the buffer

size_t * len length of line placed into buffer

Description

Start at the beginning of the kmsg buffer, with the oldest kmsg record, and copy one record into the provided buffer.

Consecutive calls will return the next available record moving towards the end of the buffer with the youngest messages.

A return value of FALSE indicates that there are no more records to read.

bool **kmsg_dump_get_buffer**(struct kmsg_dumper * dumper, bool syslog, char * buf, size_t size, size_t * len)
copy kmsg log lines

Parameters

struct kmsg_dumper * dumper registered kmsg dumper

bool syslog include the “<4>” prefixes

char * buf buffer to copy the line to

size_t size maximum size of the buffer

size_t * len length of line placed into buffer

Description

Start at the end of the kmsg buffer and fill the provided buffer with as many of the the youngest kmsg records that fit into it. If the buffer is large enough, all available kmsg records will be copied with a single call.

Consecutive calls will fill the buffer with the next block of available older records, not including the earlier retrieved ones.

A return value of FALSE indicates that there are no more records to read.

void **kmsg_dump_rewind**(struct kmsg_dumper * dumper)
reset the iterator

Parameters

struct kmsg_dumper * dumper registered kmsg dumper

Description

Reset the dumper's iterator so that `kmsg_dump_get_line()` and `kmsg_dump_get_buffer()` can be called again and used multiple times within the same dumper.dump() callback.

void **panic**(const char * fmt, ...)
halt the system

Parameters

const char * fmt The text string to print

Display a message, then perform cleanups.

This function never returns.

... variable arguments

void **add_taint**(unsigned flag, enum lockdep_ok lockdep_ok)

Parameters

unsigned flag one of the TAINT_* constants.

enum lockdep_ok lockdep_ok whether lock debugging is still OK.

Description

If something bad has gone wrong, you'll want **lockdebug_ok** = false, but for some noteworthtt-but-not-corrupting cases, it can be set to true.

bool **rcu_is_watching**(void)
see if RCU thinks that the current CPU is not idle

Parameters

void no arguments

Description

Return true if RCU is watching the running CPU, which means that this CPU can safely enter RCU read-side critical sections. In other words, if the current CPU is not in its idle loop or is in an interrupt or NMI handler, return true.

void **call_rcu**(struct rcu_head * head, rcu_callback_t func)
Queue an RCU callback for invocation after a grace period.

Parameters

struct rcu_head * head structure to be used for queueing the RCU updates.

rcu_callback_t func actual callback function to be invoked after the grace period

Description

The callback function will be invoked some time after a full grace period elapses, in other words after all pre-existing RCU read-side critical sections have completed. However, the callback function might well execute concurrently with RCU read-side critical sections that started after `call_rcu()` was invoked. RCU read-side critical sections are delimited by rcu_read_lock() and rcu_read_unlock(), and may be nested. In addition, regions of code across which interrupts, preemption, or softirqs have been disabled also serve as RCU read-side critical sections. This includes hardware interrupt handlers, softirq handlers, and NMI handlers.

Note that all CPUs must agree that the grace period extended beyond all pre-existing RCU read-side critical section. On systems with more than one CPU, this means that when "func()" is invoked, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU read-side critical section whose beginning preceded the call to `call_rcu()`. It also means that each CPU executing an RCU read-side critical section that continues beyond the start of "func()" must have executed a memory barrier after the `call_rcu()` but before the beginning of that RCU read-side critical section. Note that these guarantees include CPUs that are offline, idle, or executing in user mode, as well as CPUs that are executing in the kernel.

Furthermore, if CPU A invoked `call_rcu()` and CPU B invoked the resulting RCU callback function "func()" , then both CPU A and CPU B are guaranteed to execute a full memory barrier during the time interval between the call to `call_rcu()` and the invocation of "func()" – even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

void **`synchronize_rcu`**(void)
wait until a grace period has elapsed.

Parameters

`void` no arguments

Description

Control will return to the caller some time after a full grace period has elapsed, in other words after all currently executing RCU read-side critical sections have completed. Note, however, that upon return from `synchronize_rcu()`, the caller might well be executing concurrently with new RCU read-side critical sections that began while `synchronize_rcu()` was waiting. RCU read-side critical sections are delimited by rcu_read_lock() and rcu_read_unlock(), and may be nested. In addition, regions of code across which interrupts, preemption, or softirqs have been disabled also serve as RCU read-side critical sections. This includes hardware interrupt handlers, softirq handlers, and NMI handlers.

Note that this guarantee implies further memory-ordering guarantees. On systems with more than one CPU, when `synchronize_rcu()` returns, each CPU is guaranteed to have executed a full memory barrier since the end of its last RCU read-side critical section whose beginning preceded the call to `synchronize_rcu()`. In addition, each CPU having an RCU read-side critical section that extends beyond the return from `synchronize_rcu()` is guaranteed to have executed a full memory barrier after the beginning of `synchronize_rcu()` and before the beginning of that RCU read-side critical section. Note that these guarantees include CPUs that are offline, idle, or executing in user mode, as well as CPUs that are executing in the kernel.

Furthermore, if CPU A invoked `synchronize_rcu()`, which returned to its caller on CPU B, then both CPU A and CPU B are guaranteed to have executed a full memory barrier during the execution of `synchronize_rcu()` – even if CPU A and CPU B are the same CPU (but again only if the system has more than one CPU).

unsigned long **get_state_synchronize_rcu**(void)
Snapshot current RCU state

Parameters

void no arguments

Description

Returns a cookie that is used by a later call to `cond_synchronize_rcu()` to determine whether or not a full grace period has elapsed in the meantime.

void **cond_synchronize_rcu**(unsigned long oldstate)
Conditionally wait for an RCU grace period

Parameters

unsigned long oldstate return value from earlier call to `get_state_synchronize_rcu()`

Description

If a full RCU grace period has elapsed since the earlier call to `get_state_synchronize_rcu()`, just return. Otherwise, invoke `synchronize_rcu()` to wait for a full grace period.

Yes, this function does not take counter wrap into account. But counter wrap is harmless. If the counter wraps, we have waited for more than 2 billion grace periods (and way more on a 64-bit system!), so waiting for one additional grace period should be just fine.

void **rcu_barrier**(void)
Wait until all in-flight `call_rcu()` callbacks complete.

Parameters

void no arguments

Description

Note that this primitive does not necessarily wait for an RCU grace period to complete. For example, if there are no RCU callbacks queued anywhere in the system, then `rcu_barrier()` is within its rights to return immediately, without waiting for anything, much less an RCU grace period.

void **rcu_expedite_gp**(void)
Expedite future RCU grace periods

Parameters

void no arguments

Description

After a call to this function, future calls to `synchronize_rcu()` and friends act as the corresponding synchronize_rcu_expedited() function had instead been called.

void **rcu_unexpedite_gp**(void)
Cancel prior `rcu_expedite_gp()` invocation

Parameters

void no arguments

Description

Undo a prior call to `rcu_expedite_gp()`. If all prior calls to `rcu_expedite_gp()` are undone by a subsequent call to `rcu_unexpedite_gp()`, and if the rcu_expedited sysfs/boot parameter is not set, then all subsequent calls to `synchronize_rcu()` and friends will return to their normal non-expedited behavior.

int **rcu_read_lock_held**(void)
might we be in RCU read-side critical section?

Parameters

void no arguments

Description

If CONFIG_DEBUG_LOCK_ALLOC is selected, returns nonzero iff in an RCU read-side critical section. In absence of CONFIG_DEBUG_LOCK_ALLOC, this assumes we are in an RCU read-side critical section unless it can prove otherwise. This is useful for debug checks in functions that require that they be called within an RCU read-side critical section.

Checks debug_lockdep_rcu_enabled() to prevent false positives during boot and while lockdep is disabled.

Note that rcu_read_lock() and the matching rcu_read_unlock() must occur in the same context, for example, it is illegal to invoke rcu_read_unlock() in process context if the matching rcu_read_lock() was invoked from within an irq handler.

Note that rcu_read_lock() is disallowed if the CPU is either idle or offline from an RCU perspective, so check for those as well.

int **rcu_read_lock_bh_held**(void)
might we be in RCU-bh read-side critical section?

Parameters

void no arguments

Description

Check for bottom half being disabled, which covers both the CONFIG_PROVE_RCU and not cases. Note that if someone uses rcu_read_lock_bh(), but then later enables BH, lockdep (if enabled) will show the situation. This is useful for debug checks in functions that require that they be called within an RCU read-side critical section.

Check debug_lockdep_rcu_enabled() to prevent false positives during boot.

Note that rcu_read_lock_bh() is disallowed if the CPU is either idle or offline from an RCU perspective, so check for those as well.

void **wakeme_after_rcu**(struct rcu_head * head)
Callback function to awaken a task after grace period

Parameters

struct rcu_head * head Pointer to rcu_head member within rcu_synchronize structure

Description

Awaken the corresponding task now that a grace period has elapsed.

void **init_rcu_head_on_stack**(struct rcu_head * head)
initialize on-stack rcu_head for debugobjects

Parameters

struct rcu_head * head pointer to rcu_head structure to be initialized

Description

This function informs debugobjects of a new rcu_head structure that has been allocated as an auto variable on the stack. This function is not required for rcu_head structures that are statically defined or that are dynamically allocated on the heap. This function has no effect for !CONFIG_DEBUG_OBJECTS_RCU_HEAD kernel builds.

void **destroy_rcu_head_on_stack**(struct rcu_head * head)
destroy on-stack rcu_head for debugobjects

Parameters

struct rcu_head * head pointer to rcu_head structure to be initialized

Description

This function informs debugobjects that an on-stack rcu_head structure is about to go out of scope. As with `init_rcu_head_on_stack()`, this function is not required for rcu_head structures that are statically defined or that are dynamically allocated on the heap. Also as with `init_rcu_head_on_stack()`, this function has no effect for !CONFIG_DEBUG_OBJECTS_RCU_HEAD kernel builds.

size_t **array_size**(size_t a, size_t b)
Calculate size of 2-dimensional array.

Parameters

size_t a dimension one

size_t b dimension two

Description

Calculates size of 2-dimensional array: **a** * **b**.

Return

number of bytes needed to represent the array or SIZE_MAX on overflow.

size_t **array3_size**(size_t a, size_t b, size_t c)
Calculate size of 3-dimensional array.

Parameters

size_t a dimension one

size_t b dimension two

size_t c dimension three

Description

Calculates size of 3-dimensional array: **a** * **b** * **c**.

Return

number of bytes needed to represent the array or SIZE_MAX on overflow.

struct_size(p, member, count)
Calculate size of structure with trailing array.

Parameters

p Pointer to the structure.

member Name of the array member.

count Number of elements in the array.

Description

Calculates size of memory needed for structure **p** followed by an array of **count** number of **member** elements.

Return

number of bytes needed or SIZE_MAX on overflow.

flex_array_size(p, member, count)
Calculate size of a flexible array member within an enclosing structure.

Parameters

p Pointer to the structure.

member Name of the flexible array member.

count Number of elements in the array.

Description

Calculates size of a flexible array of **count** number of **member** elements, at the end of structure **p**.

Return

number of bytes needed or SIZE_MAX on overflow.

2.12 Device Resource Management

void * **devres_alloc_node**(dr_release_t release, size_t size, gfp_t gfp, int nid)
Allocate device resource data

Parameters

dr_release_t release Release function devres will be associated with

size_t size Allocation size

gfp_t gfp Allocation flags

int nid NUMA node

Description

Allocate devres of **size** bytes. The allocated area is zeroed, then associated with **release**. The returned pointer can be passed to other devres_*() functions.

Return

Pointer to allocated devres on success, NULL on failure.

void **devres_for_each_res**(struct device * dev, dr_release_t release, dr_match_t match, void * match_data, void (*fn)(struct device *, void *, void *), void * data)
Resource iterator

Parameters

struct device * dev Device to iterate resource from

dr_release_t release Look for resources associated with this release function

dr_match_t match Match function (optional)

void * match_data Data for the match function

void (*)(struct device *, void *, void *) fn Function to be called for each matched resource.

void * data Data for **fn**, the 3rd parameter of **fn**

Description

Call **fn** for each devres of **dev** which is associated with **release** and for which **match** returns 1.

Return

void

void **devres_free**(void * res)
Free device resource data

Parameters

void * res Pointer to devres data to free

Description

Free devres created with devres_alloc().

void **devres_add**(struct device * dev, void * res)
Register device resource

Parameters

struct device * dev Device to add resource to

void * res Resource to register

Description

Register devres **res** to **dev**. **res** should have been allocated using devres_alloc(). On driver detach, the associated release function will be invoked and devres will be freed automatically.

void * **devres_find**(struct device * dev, dr_release_t release, dr_match_t match, void * match_data)
Find device resource

Parameters

struct device * dev Device to lookup resource from

dr_release_t release Look for resources associated with this release function

dr_match_t match Match function (optional)

void * match_data Data for the match function

Description

Find the latest devres of **dev** which is associated with **release** and for which **match** returns 1. If **match** is NULL, it' s considered to match all.

Return

Pointer to found devres, NULL if not found.

void * **devres_get**(struct device * dev, void * new_res, dr_match_t match, void * match_data)
Find devres, if non-existent, add one atomically

Parameters

struct device * dev Device to lookup or add devres for

void * new_res Pointer to new initialized devres to add if not found

dr_match_t match Match function (optional)

void * match_data Data for the match function

Description

Find the latest devres of **dev** which has the same release function as **new_res** and for which **match** return 1. If found, **new_res** is freed; otherwise, **new_res** is added atomically.

Return

Pointer to found or added devres.

void * **devres_remove**(struct device * dev, dr_release_t release, dr_match_t match, void * match_data)
Find a device resource and remove it

Parameters

struct device * dev Device to find resource from

dr_release_t release Look for resources associated with this release function

dr_match_t match Match function (optional)

void * match_data Data for the match function

Description

Find the latest devres of **dev** associated with **release** and for which **match** returns 1. If **match** is NULL, it's considered to match all. If found, the resource is removed atomically and returned.

Return

Pointer to removed devres on success, NULL if not found.

int **devres_destroy**(struct device * dev, dr_release_t release, dr_match_t match, void * match_data)
Find a device resource and destroy it

Parameters

struct device * dev Device to find resource from

dr_release_t release Look for resources associated with this release function

dr_match_t match Match function (optional)

void * match_data Data for the match function

Description

Find the latest devres of **dev** associated with **release** and for which **match** returns 1. If **match** is NULL, it's considered to match all. If found, the resource is removed atomically and freed.

Note that the release function for the resource will not be called, only the devres-allocated data will be freed. The caller becomes responsible for freeing any other data.

Return

0 if devres is found and freed, -ENOENT if not found.

int **devres_release**(struct device * dev, dr_release_t release, dr_match_t match, void * match_data)
Find a device resource and destroy it, calling release

Parameters

struct device * dev Device to find resource from

dr_release_t release Look for resources associated with this release function

dr_match_t match Match function (optional)

void * match_data Data for the match function

Description

Find the latest devres of **dev** associated with **release** and for which **match** returns 1. If **match** is NULL, it's considered to match all. If found, the resource is removed atomically, the release function called and the resource freed.

Return

0 if devres is found and freed, -ENOENT if not found.

void * **devres_open_group**(struct device * dev, void * id, gfp_t gfp)
Open a new devres group

Parameters

struct device * dev Device to open devres group for

void * id Separator ID

gfp_t gfp Allocation flags

Description

Open a new devres group for **dev** with **id**. For **id**, using a pointer to an object which won't be used for another group is recommended. If **id** is NULL, address-wise unique ID is created.

Return

ID of the new group, NULL on failure.

void **devres_close_group**(struct device * dev, void * id)
Close a devres group

Parameters

struct device * dev Device to close devres group for

void * id ID of target group, can be NULL

Description

Close the group identified by **id**. If **id** is NULL, the latest open group is selected.

void **devres_remove_group**(struct device * dev, void * id)
Remove a devres group

Parameters

struct device * dev Device to remove group for

void * id ID of target group, can be NULL

Description

Remove the group identified by **id**. If **id** is NULL, the latest open group is selected. Note that removing a group doesn't affect any other resources.

int **devres_release_group**(struct device * dev, void * id)
Release resources in a devres group

Parameters

struct device * dev Device to release group for

void * id ID of target group, can be NULL

Description

Release all resources in the group identified by **id**. If **id** is NULL, the latest open group is selected. The selected group and groups properly nested inside the selected group are removed.

Return

The number of released non-group resources.

int **devm_add_action**(struct device * dev, void (*action)(void *), void * data)
add a custom action to list of managed resources

Parameters

struct device * dev Device that owns the action

void (*)(void *) action Function that should be called

void * data Pointer to data passed to **action** implementation

Description

This adds a custom action to the list of managed resources so that it gets executed as part of standard resource unwinding.

void **devm_remove_action**(struct device * dev, void (*action)(void *), void * data)
removes previously added custom action

Parameters

struct device * dev Device that owns the action

void (*)(void *) action Function implementing the action

void * data Pointer to data passed to **action** implementation

Description

Removes instance of **action** previously added by `devm_add_action()`. Both action and data should match one of the existing entries.

void **devm_release_action**(struct device * dev, void (*action)(void *), void * data)
release previously added custom action

Parameters

struct device * dev Device that owns the action

void (*)(void *) action Function implementing the action

void * data Pointer to data passed to **action** implementation

Description

Releases and removes instance of **action** previously added by `devm_add_action()`. Both action and data should match one of the existing entries.

void * **devm_kmalloc**(struct device * dev, size_t size, gfp_t gfp)
Resource-managed kmalloc

Parameters

struct device * dev Device to allocate memory for

size_t size Allocation size

gfp_t gfp Allocation gfp flags

Description

Managed kmalloc. Memory allocated with this function is automatically freed on driver detach. Like all other devres resources, guaranteed alignment is unsigned long long.

Return

Pointer to allocated memory on success, NULL on failure.

char * **devm_kstrdup**(struct device * dev, const char * s, gfp_t gfp)
Allocate resource managed space and copy an existing string into that.

Parameters

struct device * dev Device to allocate memory for

const char * s the string to duplicate

gfp_t gfp the GFP mask used in the `devm_kmalloc()` call when allocating memory

Return

Pointer to allocated string on success, NULL on failure.

const char * **devm_kstrdup_const**(struct device * dev, const char * s, gfp_t gfp)
resource managed conditional string duplication

Parameters

struct device * dev device for which to duplicate the string

const char * s the string to duplicate

gfp_t gfp the GFP mask used in the kmalloc() call when allocating memory

Description

Strings allocated by devm_kstrdup_const will be automatically freed when the associated device is detached.

Return

Source string if it is in .rodata section otherwise it falls back to devm_kstrdup.

char * **devm_kvasprintf**(struct device * dev, gfp_t gfp, const char * fmt, va_list ap)
Allocate resource managed space and format a string into that.

Parameters

struct device * dev Device to allocate memory for

gfp_t gfp the GFP mask used in the `devm_kmalloc()` call when allocating memory

const char * fmt The printf()-style format string

va_list ap Arguments for the format string

Return

Pointer to allocated string on success, NULL on failure.

char * **devm_kasprintf**(struct device * dev, gfp_t gfp, const char * fmt, ...)
Allocate resource managed space and format a string into that.

Parameters

struct device * dev Device to allocate memory for

gfp_t gfp the GFP mask used in the `devm_kmalloc()` call when allocating memory

const char * fmt The printf()-style format string

... Arguments for the format string

Return

Pointer to allocated string on success, NULL on failure.

void **devm_kfree**(struct device * dev, const void * p)
Resource-managed kfree

Parameters

struct device * dev Device this memory belongs to

const void * p Memory to free

Description

Free memory allocated with `devm_kmalloc()`.

void * **devm_kmemdup**(struct device * dev, const void * src, size_t len, gfp_t gfp)
Resource-managed kmemdup

Parameters

struct device * dev Device this memory belongs to

const void * src Memory region to duplicate

size_t len Memory region length

gfp_t gfp GFP mask to use

Description

Duplicate region of a memory using resource managed kmalloc

unsigned long **devm_get_free_pages**(struct device * dev, gfp_t gfp_mask, unsigned int order)
Resource-managed __get_free_pages

Parameters

struct device * dev Device to allocate memory for

gfp_t gfp_mask Allocation gfp flags

unsigned int order Allocation size is (1 << order) pages

Description

Managed get_free_pages. Memory allocated with this function is automatically freed on driver detach.

Return

Address of allocated memory on success, 0 on failure.

void **devm_free_pages**(struct device * dev, unsigned long addr)
Resource-managed free_pages

Parameters

struct device * dev Device this memory belongs to

unsigned long addr Memory to free

Description

Free memory allocated with `devm_get_free_pages()`. Unlike free_pages, there is no need to supply the **order**.

void __percpu * **__devm_alloc_percpu**(struct device * dev, size_t size, size_t align)
Resource-managed alloc_percpu

Parameters

struct device * dev Device to allocate per-cpu memory for

size_t size Size of per-cpu memory to allocate

size_t align Alignment of per-cpu memory to allocate

Description

Managed alloc_percpu. Per-cpu memory allocated with this function is automatically freed on driver detach.

Return

Pointer to allocated memory on success, NULL on failure.

void **devm_free_percpu**(struct device * dev, void __percpu * pdata)
Resource-managed free_percpu

Parameters

struct device * dev Device this memory belongs to

void __percpu * pdata Per-cpu memory to free

Description

Free memory allocated with `devm_alloc_percpu()`.

CHAPTER

THREE

DEVICE DRIVERS INFRASTRUCTURE

3.1 The Basic Device Driver-Model Structures

struct **subsys_interface**
interfaces to device functions

Definition

```
struct subsys_interface {
  const char *name;
  struct bus_type *subsys;
  struct list_head node;
  int (*add_dev)(struct device *dev, struct subsys_interface *sif);
  void (*remove_dev)(struct device *dev, struct subsys_interface *sif);
};
```

Members

name name of the device function

subsys subsytem of the devices to attach to

node the list of functions registered at the subsystem

add_dev device hookup to device function handler

remove_dev device hookup to device function handler

Description

Simple interfaces attached to a subsystem. Multiple interfaces can attach to a subsystem and its devices. Unlike drivers, they do not exclusively claim or control devices. Interfaces usually represent a specific functionality of a subsystem/class of devices.

devm_alloc_percpu(dev, type)
Resource-managed alloc_percpu

Parameters

dev Device to allocate per-cpu memory for

type Type to allocate per-cpu memory for

Description

Managed alloc_percpu. Per-cpu memory allocated with this function is automatically freed on driver detach.

Return

Pointer to allocated memory on success, NULL on failure.

struct **device_connection**
Device Connection Descriptor

Definition

```
struct device_connection {
  struct fwnode_handle    *fwnode;
  const char              *endpoint[2];
  const char              *id;
  struct list_head        list;
};
```

Members

fwnode The device node of the connected device

endpoint The names of the two devices connected together

id Unique identifier for the connection

list List head, private, for internal use only

NOTE

fwnode is not used together with **endpoint**. **fwnode** is used when platform firmware defines the connection. When the connection is registered with `device_connection_add()` **endpoint** is used instead.

void **device_connections_add**(struct device_connection * cons)
Add multiple device connections at once

Parameters

struct device_connection * cons Zero terminated array of device connection descriptors

void **device_connections_remove**(struct device_connection * cons)
Remove multiple device connections at once

Parameters

struct device_connection * cons Zero terminated array of device connection descriptors

enum **device_link_state**
Device link states.

Constants

DL_STATE_NONE The presence of the drivers is not being tracked.

DL_STATE_DORMANT None of the supplier/consumer drivers is present.

DL_STATE_AVAILABLE The supplier driver is present, but the consumer is not.

DL_STATE_CONSUMER_PROBE The consumer is probing (supplier driver present).

DL_STATE_ACTIVE Both the supplier and consumer drivers are present.

DL_STATE_SUPPLIER_UNBIND The supplier driver is unbinding.

struct **device_link**
Device link representation.

Definition

```
struct device_link {
  struct device *supplier;
  struct list_head s_node;
  struct device *consumer;
  struct list_head c_node;
  enum device_link_state status;
  u32 flags;
  refcount_t rpm_active;
  struct kref kref;
#ifdef CONFIG_SRCU;
  struct rcu_head rcu_head;
#endif;
  bool supplier_preactivated;
};
```

Members

supplier The device on the supplier end of the link.

s_node Hook to the supplier device's list of links to consumers.

consumer The device on the consumer end of the link.

c_node Hook to the consumer device's list of links to suppliers.

status The state of the link (with respect to the presence of drivers).

flags Link flags.

rpm_active Whether or not the consumer device is runtime-PM-active.

kref Count repeated addition of the same link.

rcu_head An RCU head to use for deferred execution of SRCU callbacks.

supplier_preactivated Supplier has been made active before consumer probe.

enum **dl_dev_state**
Device driver presence tracking information.

Constants

DL_DEV_NO_DRIVER There is no driver attached to the device.

DL_DEV_PROBING A driver is probing.

DL_DEV_DRIVER_BOUND The driver has been bound to the device.

DL_DEV_UNBINDING The driver is unbinding from the device.

struct **dev_links_info**
Device data related to device links.

Definition

```
struct dev_links_info {
  struct list_head suppliers;
```

(continues on next page)

(continued from previous page)

```
  struct list_head consumers;
  struct list_head needs_suppliers;
  struct list_head defer_sync;
  bool need_for_probe;
  enum dl_dev_state status;
};
```

Members

suppliers List of links to supplier devices.

consumers List of links to consumer devices.

needs_suppliers Hook to global list of devices waiting for suppliers.

defer_sync Hook to global list of devices that have deferred sync_state.

need_for_probe If needs_suppliers is on a list, this indicates if the suppliers are needed for probe or not.

status Driver status information.

struct **device**
The basic device structure

Definition

```
struct device {
  struct kobject kobj;
  struct device           *parent;
  struct device_private   *p;
  const char              *init_name;
  const struct device_type *type;
  struct bus_type *bus;
  struct device_driver *driver;
  void *platform_data;
  void *driver_data;
#ifdef CONFIG_PROVE_LOCKING;
  struct mutex            lockdep_mutex;
#endif;
  struct mutex            mutex;
  struct dev_links_info   links;
  struct dev_pm_info      power;
  struct dev_pm_domain    *pm_domain;
#ifdef CONFIG_GENERIC_MSI_IRQ_DOMAIN;
  struct irq_domain       *msi_domain;
#endif;
#ifdef CONFIG_PINCTRL;
  struct dev_pin_info     *pins;
#endif;
#ifdef CONFIG_GENERIC_MSI_IRQ;
  struct list_head        msi_list;
#endif;
  const struct dma_map_ops *dma_ops;
  u64 *dma_mask;
  u64 coherent_dma_mask;
  u64 bus_dma_limit;
  unsigned long   dma_pfn_offset;
```

(continues on next page)

(continued from previous page)

```
  struct device_dma_parameters *dma_parms;
  struct list_head        dma_pools;
#ifdef CONFIG_DMA_DECLARE_COHERENT;
  struct dma_coherent_mem *dma_mem;
#endif;
#ifdef CONFIG_DMA_CMA;
  struct cma *cma_area;
#endif;
  struct dev_archdata     archdata;
  struct device_node      *of_node;
  struct fwnode_handle    *fwnode;
#ifdef CONFIG_NUMA;
  int numa_node;
#endif;
  dev_t devt;
  u32 id;
  spinlock_t devres_lock;
  struct list_head        devres_head;
  struct class            *class;
  const struct attribute_group **groups;
  void (*release)(struct device *dev);
  struct iommu_group      *iommu_group;
  struct dev_iommu        *iommu;
  bool offline_disabled:1;
  bool offline:1;
  bool of_node_reused:1;
  bool state_synced:1;
#if defined(CONFIG_ARCH_HAS_SYNC_DMA_FOR_DEVICE) ||     defined(CONFIG_
↪ARCH_HAS_SYNC_DMA_FOR_CPU) ||     defined(CONFIG_ARCH_HAS_SYNC_DMA_FOR_
↪CPU_ALL);
  bool dma_coherent:1;
#endif;
};
```

Members

kobj A top-level, abstract class from which other classes are derived.

parent The device's "parent" device, the device to which it is attached. In most cases, a parent device is some sort of bus or host controller. If parent is NULL, the device, is a top-level device, which is not usually what you want.

p Holds the private data of the driver core portions of the device. See the comment of the struct device_private for detail.

init_name Initial name of the device.

type The type of device. This identifies the device type and carries type-specific information.

bus Type of bus device is on.

driver Which driver has allocated this

platform_data Platform data specific to the device.

driver_data Private pointer for driver specific info.

lockdep_mutex An optional debug lock that a subsystem can use as a peer lock to gain localized lockdep coverage of the device_lock.

mutex Mutex to synchronize calls to its driver.

links Links to suppliers and consumers of this device.

power For device power management. See Documentation/driver-api/pm/devices.rst for details.

pm_domain Provide callbacks that are executed during system suspend, hibernation, system resume and during runtime PM transitions along with subsystem-level and driver-level callbacks.

msi_domain The generic MSI domain this device is using.

pins For device pin management. See Documentation/driver-api/pinctl.rst for details.

msi_list Hosts MSI descriptors

dma_ops DMA mapping operations for this device.

dma_mask Dma mask (if dma’ble device).

coherent_dma_mask Like dma_mask, but for alloc_coherent mapping as not all hardware supports 64-bit addresses for consistent allocations such descriptors.

bus_dma_limit Limit of an upstream bridge or bus which imposes a smaller DMA limit than the device itself supports.

dma_pfn_offset offset of DMA memory range relatively of RAM

dma_parms A low level driver may set these to teach IOMMU code about segment limitations.

dma_pools Dma pools (if dma’ble device).

dma_mem Internal for coherent mem override.

cma_area Contiguous memory area for dma allocations

archdata For arch-specific additions.

of_node Associated device tree node.

fwnode Associated device node supplied by platform firmware.

numa_node NUMA node this device is close to.

devt For creating the sysfs “dev”.

id device instance

devres_lock Spinlock to protect the resource of the device.

devres_head The resources list of the device.

class The class of the device.

groups Optional attribute groups.

release Callback to free the device after all references have gone away. This should be set by the allocator of the device (i.e. the bus driver that discovered the device).

iommu_group IOMMU group the device belongs to.

iommu Per device generic IOMMU runtime data

offline_disabled If set, the device is permanently online.

offline Set after successful invocation of bus type's .offline().

of_node_reused Set if the device-tree node is shared with an ancestor device.

state_synced The hardware state of this device has been synced to match the software state of this device by calling the driver/bus sync_state() callback.

dma_coherent this particular device is dma coherent, even if the architecture supports non-coherent devices.

Example

For devices on custom boards, as typical of embedded and SOC based hardware, Linux often uses platform_data to point to board-specific structures describing devices and how they are wired. That can include what ports are available, chip variants, which GPIO pins act in what additional roles, and so on. This shrinks the "Board Support Packages" (BSPs) and minimizes board-specific #ifdefs in drivers.

Description

At the lowest level, every device in a Linux system is represented by an instance of struct device. The device structure contains the information that the device model core needs to model the system. Most subsystems, however, track additional information about the devices they host. As a result, it is rare for devices to be represented by bare device structures; instead, that structure, like kobject structures, is usually embedded within a higher-level representation of the device.

bool **device_iommu_mapped**(struct device * dev)
Returns true when the device DMA is translated by an IOMMU

Parameters

struct device * dev Device to perform the check on

3.2 Device Drivers Base

void **driver_init**(void)
initialize driver model.

Parameters

void no arguments

Description

Call the driver model init functions to initialize their subsystems. Called early from init/main.c.

int **driver_for_each_device**(struct device_driver * drv, struct device * start, void * data, int (*fn)(struct device *, void *))
Iterator for devices bound to a driver.

Parameters

struct device_driver * drv Driver we're iterating.

struct device * start Device to begin with

void * data Data to pass to the callback.

int (*)(struct device *, void *) fn Function to call for each device.

Description

Iterate over the **drv**'s list of devices calling **fn** for each one.

struct device * **driver_find_device**(struct device_driver * drv, struct device * start, const void * data, int (*match)(struct device *dev, const void *data))
device iterator for locating a particular device.

Parameters

struct device_driver * drv The device's driver

struct device * start Device to begin with

const void * data Data to pass to match function

int (*)(struct device *dev, const void *data) match Callback function to check device

Description

This is similar to the `driver_for_each_device()` function above, but it returns a reference to a device that is 'found' for later use, as determined by the **match** callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

int **driver_create_file**(struct device_driver * drv, const struct driver_attribute * attr)
create sysfs file for driver.

Parameters

struct device_driver * drv driver.

const struct driver_attribute * attr driver attribute descriptor.

void **driver_remove_file**(struct device_driver * drv, const struct driver_attribute * attr)
remove sysfs file for driver.

Parameters

struct device_driver * drv driver.

const struct driver_attribute * attr driver attribute descriptor.

int **driver_register**(struct device_driver * drv)
register driver with bus

Parameters

struct device_driver * drv driver to register

Description

We pass off most of the work to the bus_add_driver() call, since most of the things we have to do deal with the bus structures.

void **driver_unregister**(struct device_driver * drv)
remove driver from system.

Parameters

struct device_driver * drv driver.

Description

Again, we pass off most of the work to the bus-level call.

struct device_driver * **driver_find**(const char * name, struct bus_type * bus)
locate driver on a bus by its name.

Parameters

const char * name name of the driver.

struct bus_type * bus bus to scan for the driver.

Description

Call `kset_find_obj()` to iterate over list of drivers on a bus to find driver by name. Return driver if found.

This routine provides no locking to prevent the driver it returns from being unregistered or unloaded while the caller is using it. The caller is responsible for preventing this.

struct device_link * **device_link_add**(struct device * consumer, struct device * supplier, u32 flags)
Create a link between two devices.

Parameters

struct device * consumer Consumer end of the link.

struct device * supplier Supplier end of the link.

u32 flags Link flags.

Description

The caller is responsible for the proper synchronization of the link creation with runtime PM. First, setting the DL_FLAG_PM_RUNTIME flag will cause the runtime PM framework to take the link into account. Second, if the DL_FLAG_RPM_ACTIVE flag is set in addition to it, the supplier devices will be forced into the active metastate and reference-counted upon the creation of the

link. If DL_FLAG_PM_RUNTIME is not set, DL_FLAG_RPM_ACTIVE will be ignored.

If DL_FLAG_STATELESS is set in **flags**, the caller of this function is expected to release the link returned by it directly with the help of either `device_link_del()` or `device_link_remove()`.

If that flag is not set, however, the caller of this function is handing the management of the link over to the driver core entirely and its return value can only be used to check whether or not the link is present. In that case, the DL_FLAG_AUTOREMOVE_CONSUMER and DL_FLAG_AUTOREMOVE_SUPPLIER device link flags can be used to indicate to the driver core when the link can be safely deleted. Namely, setting one of them in **flags** indicates to the driver core that the link is not going to be used (by the given caller of this function) after unbinding the consumer or supplier driver, respectively, from its device, so the link can be deleted at that point. If none of them is set, the link will be maintained until one of the devices pointed to by it (either the consumer or the supplier) is unregistered.

Also, if DL_FLAG_STATELESS, DL_FLAG_AUTOREMOVE_CONSUMER and DL_FLAG_AUTOREMOVE_SUPPLIER are not set in **flags** (that is, a persistent managed device link is being added), the DL_FLAG_AUTOPROBE_CONSUMER flag can be used to request the driver core to automaticall probe for a consmer driver after successfully binding a driver to the supplier device.

The combination of DL_FLAG_STATELESS and one of DL_FLAG_AUTOREMOVE_CONSUMER, DL_FLAG_AUTOREMOVE_SUPPLIER, or DL_FLAG_AUTOPROBE_CONSUMER set in **flags** at the same time is invalid and will cause NULL to be returned upfront. However, if a device link between the given **consumer** and **supplier** pair exists already when this function is called for them, the existing link will be returned regardless of its current type and status (the link' s flags may be modified then). The caller of this function is then expected to treat the link as though it has just been created, so (in particular) if DL_FLAG_STATELESS was passed in **flags**, the link needs to be released explicitly when not needed any more (as stated above).

A side effect of the link creation is re-ordering of dpm_list and the devices_kset list by moving the consumer device and all devices depending on it to the ends of these lists (that does not happen to devices that have not been registered when this function is called).

The supplier device is required to be registered when this function is called and NULL will be returned if that is not the case. The consumer device need not be registered, however.

void **device_link_del**(struct device_link * link)
Delete a stateless link between two devices.

Parameters

struct device_link * link Device link to delete.

Description

The caller must ensure proper synchronization of this function with runtime PM. If the link was added multiple times, it needs to be deleted as often. Care is

required for hotplugged devices: Their links are purged on removal and calling `device_link_del()` is then no longer allowed.

void **device_link_remove**(void * consumer, struct device * supplier)
Delete a stateless link between two devices.

Parameters

void * consumer Consumer end of the link.

struct device * supplier Supplier end of the link.

Description

The caller must ensure proper synchronization of this function with runtime PM.

const char * **dev_driver_string**(const struct device * dev)
Return a device's driver name, if at all possible

Parameters

const struct device * dev struct device to get the name of

Description

Will return the device's driver's name if it is bound to a device. If the device is not bound to a driver, it will return the name of the bus it is attached to. If it is not attached to a bus either, an empty string will be returned.

int **devm_device_add_group**(struct device * dev, const struct attribute_group * grp)
given a device, create a managed attribute group

Parameters

struct device * dev The device to create the group for

const struct attribute_group * grp The attribute group to create

Description

This function creates a group for the first time. It will explicitly warn and error if any of the attribute files being created already exist.

Returns 0 on success or error code on failure.

void **devm_device_remove_group**(struct device * dev, const struct attribute_group * grp)

Parameters

struct device * dev device to remove the group from

const struct attribute_group * grp group to remove

Description

This function removes a group of attributes from a device. The attributes previously have to have been created for this group, otherwise it will fail.

int **devm_device_add_groups**(struct device * dev, const struct attribute_group ** groups)
create a bunch of managed attribute groups

Parameters

struct device * dev The device to create the group for

const struct attribute_group ** groups The attribute groups to create, NULL terminated

Description

This function creates a bunch of managed attribute groups. If an error occurs when creating a group, all previously created groups will be removed, unwinding everything back to the original state when this function was called. It will explicitly warn and error if any of the attribute files being created already exist.

Returns 0 on success or error code from sysfs_create_group on failure.

void **devm_device_remove_groups**(struct device * dev, const struct attribute_group ** groups)
remove a list of managed groups

Parameters

struct device * dev The device for the groups to be removed from

const struct attribute_group ** groups NULL terminated list of groups to be removed

Description

If groups is not NULL, remove the specified groups from the device.

int **device_create_file**(struct device * dev, const struct device_attribute * attr)
create sysfs attribute file for device.

Parameters

struct device * dev device.

const struct device_attribute * attr device attribute descriptor.

void **device_remove_file**(struct device * dev, const struct device_attribute * attr)
remove sysfs attribute file.

Parameters

struct device * dev device.

const struct device_attribute * attr device attribute descriptor.

bool **device_remove_file_self**(struct device * dev, const struct device_attribute * attr)
remove sysfs attribute file from its own method.

Parameters

struct device * dev device.

const struct device_attribute * attr device attribute descriptor.

Description

See kernfs_remove_self() for details.

int **device_create_bin_file**(struct device * dev, const struct bin_attribute * attr)
create sysfs binary attribute file for device.

Parameters

struct device * dev device.

const struct bin_attribute * attr device binary attribute descriptor.

void **device_remove_bin_file**(struct device * dev, const struct bin_attribute * attr)
remove sysfs binary attribute file

Parameters

struct device * dev device.

const struct bin_attribute * attr device binary attribute descriptor.

void **device_initialize**(struct device * dev)
init device structure.

Parameters

struct device * dev device.

Description

This prepares the device for use by other layers by initializing its fields. It is the first half of `device_register()`, if called by that function, though it can also be called separately, so one may use **dev**' s fields. In particular, `get_device()`/`put_device()` may be used for reference counting of **dev** after calling this function.

All fields in **dev** must be initialized by the caller to 0, except for those explicitly set to some other value. The simplest approach is to use kzalloc() to allocate the structure containing **dev**.

NOTE

Use `put_device()` to give up your reference instead of freeing **dev** directly once you have called this function.

int **dev_set_name**(struct device * dev, const char * fmt, ...)
set a device name

Parameters

struct device * dev device

const char * fmt format string for the device' s name

... variable arguments

int **device_add**(struct device * dev)
add device to device hierarchy.

Parameters

struct device * dev device.

Description

This is part 2 of `device_register()`, though may be called separately _iff_ `device_initialize()` has been called separately.

This adds **dev** to the kobject hierarchy via `kobject_add()`, adds it to the global and sibling lists for the device, then adds it to the other relevant subsystems of the driver model.

Do not call this routine or `device_register()` more than once for any device structure. The driver model core is not designed to work with devices that get unregistered and then spring back to life. (Among other things, it's very hard to guarantee that all references to the previous incarnation of **dev** have been dropped.) Allocate and register a fresh new struct device instead.

Rule of thumb is: if `device_add()` succeeds, you should call `device_del()` when you want to get rid of it. If `device_add()` has not succeeded, use only `put_device()` to drop the reference count.

NOTE

Never directly free **dev** after calling this function, even if it returned an error! Always use `put_device()` to give up your reference instead.

int **device_register**(struct device * dev)
register a device with the system.

Parameters

struct device * dev pointer to the device structure

Description

This happens in two clean steps - initialize the device and add it to the system. The two steps can be called separately, but this is the easiest and most common. I.e. you should only call the two helpers separately if have a clearly defined need to use and refcount the device before it is added to the hierarchy.

For more information, see the kerneldoc for `device_initialize()` and `device_add()`.

NOTE

Never directly free **dev** after calling this function, even if it returned an error! Always use `put_device()` to give up the reference initialized in this function instead.

struct device * **get_device**(struct device * dev)
increment reference count for device.

Parameters

struct device * dev device.

Description

This simply forwards the call to `kobject_get()`, though we do take care to provide for the case that we get a NULL pointer passed in.

void **put_device**(struct device * dev)
decrement reference count.

Parameters

struct device * dev device in question.

void **device_del**(struct device * dev)
delete device from system.

Parameters

struct device * dev device.

Description

This is the first part of the device unregistration sequence. This removes the device from the lists we control from here, has it removed from the other driver model subsystems it was added to in `device_add()`, and removes it from the kobject hierarchy.

NOTE

this should be called manually _iff_ `device_add()` was also called manually.

void **device_unregister**(struct device * dev)
unregister device from system.

Parameters

struct device * dev device going away.

Description

We do this in two parts, like we do `device_register()`. First, we remove it from all the subsystems with `device_del()`, then we decrement the reference count via `put_device()`. If that is the final reference count, the device will be cleaned up via device_release() above. Otherwise, the structure will stick around until the final reference to the device is dropped.

int **device_for_each_child**(struct device * parent, void * data, int (*fn)(struct device *dev, void *data))
device child iterator.

Parameters

struct device * parent parent struct device.

void * data data for the callback.

int (*)(struct device *dev, void *data) fn function to be called for each device.

Description

Iterate over **parent**’ s child devices, and call **fn** for each, passing it **data**.

We check the return of **fn** each time. If it returns anything other than 0, we break out and return that value.

int **device_for_each_child_reverse**(struct device * parent, void * data, int (*fn)(struct device *dev, void *data))
device child iterator in reversed order.

Parameters

struct device * parent parent struct device.

void * data data for the callback.

int (*)(struct device *dev, void *data) fn function to be called for each device.

Description

Iterate over **parent**'s child devices, and call **fn** for each, passing it **data**.

We check the return of **fn** each time. If it returns anything other than 0, we break out and return that value.

struct device * **device_find_child**(struct device * parent, void * data, int (*match)(struct device *dev, void *data))
device iterator for locating a particular device.

Parameters

struct device * parent parent struct device

void * data Data to pass to match function

int (*)(struct device *dev, void *data) match Callback function to check device

Description

This is similar to the `device_for_each_child()` function above, but it returns a reference to a device that is 'found' for later use, as determined by the **match** callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero and a reference to the current device can be obtained, this function will return to the caller and not iterate over any more devices.

NOTE

you will need to drop the reference with `put_device()` after use.

struct device * **device_find_child_by_name**(struct device * parent, const char * name)
device iterator for locating a child device.

Parameters

struct device * parent parent struct device

const char * name name of the child device

Description

This is similar to the `device_find_child()` function above, but it returns a reference to a device that has the name **name**.

NOTE

you will need to drop the reference with `put_device()` after use.

struct device * **__root_device_register**(const char * name, struct module * owner)
allocate and register a root device

Parameters

const char * name root device name

struct module * owner owner module of the root device, usually THIS_MODULE

Description

This function allocates a root device and registers it using `device_register()`. In order to free the returned device, use `root_device_unregister()`.

Root devices are dummy devices which allow other devices to be grouped under /sys/devices. Use this function to allocate a root device and then use it as the parent of any device which should appear under /sys/devices/{name}

The /sys/devices/{name} directory will also contain a 'module' symlink which points to the **owner** directory in sysfs.

Returns `struct device` pointer on success, or ERR_PTR() on error.

Note

You probably want to use root_device_register().

void **root_device_unregister**(struct device * dev)
unregister and free a root device

Parameters

struct device * dev device going away

Description

This function unregisters and cleans up a device that was created by root_device_register().

struct device * **device_create**(struct class * class, struct device * parent, dev_t devt, void * drvdata, const char * fmt, ...)
creates a device and registers it with sysfs

Parameters

struct class * class pointer to the struct class that this device should be registered to

struct device * parent pointer to the parent struct device of this new device, if any

dev_t devt the dev_t for the char device to be added

void * drvdata the data to be added to the device for callbacks

const char * fmt string for the device's name

... variable arguments

Description

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class.

A "dev" file will be created, showing the dev_t for the device, if the dev_t is not 0,0. If a pointer to a parent struct device is passed in, the newly created struct

device will be a child of that device in sysfs. The pointer to the struct device will be returned from the call. Any further sysfs files that might be required can be created using this pointer.

Returns `struct device` pointer on success, or ERR_PTR() on error.

Note

the struct class passed to this function must have previously been created with a call to class_create().

struct device * **device_create_with_groups**(struct class * class, struct device * parent, dev_t devt, void * drvdata, const struct attribute_group ** groups, const char * fmt, ...)
creates a device and registers it with sysfs

Parameters

struct class * class pointer to the struct class that this device should be registered to

struct device * parent pointer to the parent struct device of this new device, if any

dev_t devt the dev_t for the char device to be added

void * drvdata the data to be added to the device for callbacks

const struct attribute_group ** groups NULL-terminated list of attribute groups to be created

const char * fmt string for the device's name

... variable arguments

Description

This function can be used by char device classes. A struct device will be created in sysfs, registered to the specified class. Additional attributes specified in the groups parameter will also be created automatically.

A "dev" file will be created, showing the dev_t for the device, if the dev_t is not 0,0. If a pointer to a parent struct device is passed in, the newly created struct device will be a child of that device in sysfs. The pointer to the struct device will be returned from the call. Any further sysfs files that might be required can be created using this pointer.

Returns `struct device` pointer on success, or ERR_PTR() on error.

Note

the struct class passed to this function must have previously been created with a call to class_create().

void **device_destroy**(struct class * class, dev_t devt)
removes a device that was created with `device_create()`

Parameters

struct class * class pointer to the struct class that this device was registered with

dev_t devt the dev_t of the device that was previously registered

Description

This call unregisters and cleans up a device that was created with a call to `device_create()`.

int **device_rename**(struct device * dev, const char * new_name)
renames a device

Parameters

struct device * dev the pointer to the struct device to be renamed

const char * new_name the new name of the device

Description

It is the responsibility of the caller to provide mutual exclusion between two different calls of device_rename on the same device to ensure that new_name is valid and won't conflict with other devices.

Renaming devices is racy at many levels, symlinks and other stuff are not replaced atomically, and you get a "move" uevent, but it's not easy to connect the event to the old and new device. Device nodes are not renamed at all, there isn't even support for that in the kernel now.

In the meantime, during renaming, your target name might be taken by another driver, creating conflicts. Or the old name is taken directly after you renamed it – then you get events for the same DEVPATH, before you even see the "move" event. It's just a mess, and nothing new should ever rely on kernel device renaming. Besides that, it's not even implemented now for other things than (driver-core wise very simple) network devices.

We are currently about to change network renaming in udev to completely disallow renaming of devices in the same namespace as the kernel uses, because we can't solve the problems properly, that arise with swapping names of multiple interfaces without races. Means, renaming of eth[0-9]* will only be allowed to some other name than eth[0-9]*, for the aforementioned reasons.

Make up a "real" name in the driver before you register anything, or add some other attributes for userspace to find the device, or use udev to add symlinks – but never rename kernel devices later, it's a complete mess. We don't even want to get into that and try to implement the missing pieces in the core. We really have other pieces to fix in the driver core mess. :)

Note

Don't call this function. Currently, the networking layer calls this function, but that will change. The following text from Kay Sievers offers some insight:

int **device_move**(struct device * dev, struct device * new_parent, enum dpm_order dpm_order)
moves a device to a new parent

Parameters

struct device * dev the pointer to the struct device to be moved

struct device * new_parent the new parent of the device (can be NULL)

enum dpm_order dpm_order how to reorder the dpm_list

int **device_change_owner**(struct device * dev, kuid_t kuid, kgid_t kgid)
change the owner of an existing device.

Parameters

struct device * dev device.

kuid_t kuid new owner's kuid

kgid_t kgid new owner's kgid

Description

This changes the owner of **dev** and its corresponding sysfs entries to **kuid/kgid**. This function closely mirrors how **dev** was added via driver core.

Returns 0 on success or error code on failure.

void **set_primary_fwnode**(struct device * dev, struct fwnode_handle * fwnode)
Change the primary firmware node of a given device.

Parameters

struct device * dev Device to handle.

struct fwnode_handle * fwnode New primary firmware node of the device.

Description

Set the device's firmware node pointer to **fwnode**, but if a secondary firmware node of the device is present, preserve it.

void **set_secondary_fwnode**(struct device * dev, struct fwnode_handle * fwnode)
Change the secondary firmware node of a given device.

Parameters

struct device * dev Device to handle.

struct fwnode_handle * fwnode New secondary firmware node of the device.

Description

If a primary firmware node of the device is present, set its secondary pointer to **fwnode**. Otherwise, set the device's firmware node pointer to **fwnode**.

void **device_set_of_node_from_dev**(struct device * dev, const struct device * dev2)
reuse device-tree node of another device

Parameters

struct device * dev device whose device-tree node is being set

const struct device * dev2 device whose device-tree node is being reused

Description

Takes another reference to the new device-tree node after first dropping any reference held to the old node.

void **register_syscore_ops**(struct syscore_ops * ops)
Register a set of system core operations.

Parameters

struct syscore_ops * ops System core operations to register.

void **unregister_syscore_ops**(struct syscore_ops * ops)
Unregister a set of system core operations.

Parameters

struct syscore_ops * ops System core operations to unregister.

int **syscore_suspend**(void)
Execute all the registered system core suspend callbacks.

Parameters

void no arguments

Description

This function is executed with one CPU on-line and disabled interrupts.

void **syscore_resume**(void)
Execute all the registered system core resume callbacks.

Parameters

void no arguments

Description

This function is executed with one CPU on-line and disabled interrupts.

struct class * **__class_create**(struct module * owner, const char * name,
struct lock_class_key * key)
create a struct class structure

Parameters

struct module * owner pointer to the module that is to “own” this struct class

const char * name pointer to a string for the name of this class.

struct lock_class_key * key the lock_class_key for this class; used by mutex lock debugging

Description

This is used to create a struct class pointer that can then be used in calls to `device_create()`.

Returns `struct class` pointer on success, or ERR_PTR() on error.

Note, the pointer created here is to be destroyed when finished by making a call to `class_destroy()`.

void **class_destroy**(struct class * cls)
destroys a struct class structure

Parameters

struct class * cls pointer to the struct class that is to be destroyed

Description

Note, the pointer to be destroyed must have been created with a call to class_create().

void **class_dev_iter_init**(struct class_dev_iter * iter, struct class * class, struct device * start, const struct device_type * type)
initialize class device iterator

Parameters

struct class_dev_iter * iter class iterator to initialize

struct class * class the class we wanna iterate over

struct device * start the device to start iterating from, if any

const struct device_type * type device_type of the devices to iterate over, NULL for all

Description

Initialize class iterator **iter** such that it iterates over devices of **class**. If **start** is set, the list iteration will start there, otherwise if it is NULL, the iteration starts at the beginning of the list.

struct device * **class_dev_iter_next**(struct class_dev_iter * iter)
iterate to the next device

Parameters

struct class_dev_iter * iter class iterator to proceed

Description

Proceed **iter** to the next device and return it. Returns NULL if iteration is complete.

The returned device is referenced and won't be released till iterator is proceed to the next device or exited. The caller is free to do whatever it wants to do with the device including calling back into class code.

void **class_dev_iter_exit**(struct class_dev_iter * iter)
finish iteration

Parameters

struct class_dev_iter * iter class iterator to finish

Description

Finish an iteration. Always call this function after iteration is complete whether the iteration ran till the end or not.

int **class_for_each_device**(struct class * class, struct device * start, void * data, int (*fn)(struct device *, void *))
device iterator

Parameters

struct class * class the class we're iterating

struct device * start the device to start with in the list, if any.

void * data data for the callback

int (*)(struct device *, void *) fn function to be called for each device

Description

Iterate over **class**'s list of devices, and call **fn** for each, passing it **data**. If **start** is set, the list iteration will start there, otherwise if it is NULL, the iteration starts at the beginning of the list.

We check the return of **fn** each time. If it returns anything other than 0, we break out and return that value.

fn is allowed to do anything including calling back into class code. There's no locking restriction.

struct device * **class_find_device**(struct class * class, struct device * start, const void * data, int (*match)(struct device *, const void *))
device iterator for locating a particular device

Parameters

struct class * class the class we're iterating

struct device * start Device to begin with

const void * data data for the match function

int (*)(struct device *, const void *) match function to check device

Description

This is similar to the class_for_each_dev() function above, but it returns a reference to a device that is 'found' for later use, as determined by the **match** callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

Note, you will need to drop the reference with `put_device()` after use.

match is allowed to do anything including calling back into class code. There's no locking restriction.

struct class_compat * **class_compat_register**(const char * name)
register a compatibility class

Parameters

const char * name the name of the class

Description

Compatibility class are meant as a temporary user-space compatibility workaround when converting a family of class devices to a bus devices.

void **class_compat_unregister**(struct class_compat * cls)
unregister a compatibility class

Parameters

struct class_compat * cls the class to unregister

int **class_compat_create_link**(struct class_compat * cls, struct device * dev, struct device * device_link)
create a compatibility class device link to a bus device

Parameters

struct class_compat * cls the compatibility class

struct device * dev the target bus device

struct device * device_link an optional device to which a "device" link should be created

void **class_compat_remove_link**(struct class_compat * cls, struct device * dev, struct device * device_link)
remove a compatibility class device link to a bus device

Parameters

struct class_compat * cls the compatibility class

struct device * dev the target bus device

struct device * device_link an optional device to which a "device" link was previously created

struct **node_access_nodes**
Access class device to hold user visible relationships to other nodes.

Definition

```
struct node_access_nodes {
  struct device           dev;
  struct list_head        list_node;
  unsigned access;
#ifdef CONFIG_HMEM_REPORTING;
  struct node_hmem_attrs  hmem_attrs;
#endif;
};
```

Members

dev Device for this memory access class

list_node List element in the node's access list

access The access class rank

hmem_attrs Heterogeneous memory performance attributes

void **node_set_perf_attrs**(unsigned int nid, struct node_hmem_attrs * hmem_attrs, unsigned access)
Set the performance values for given access class

Parameters

unsigned int nid Node identifier to be set

struct node_hmem_attrs * hmem_attrs Heterogeneous memory performance attributes

unsigned access The access class the for the given attributes

struct **node_cache_info**
Internal tracking for memory node caches

Definition

```
struct node_cache_info {
  struct device dev;
  struct list_head node;
  struct node_cache_attrs cache_attrs;
};
```

Members

dev Device represeting the cache level

node List element for tracking in the node

cache_attrs Attributes for this cache level

void **node_add_cache**(unsigned int nid, struct node_cache_attrs * cache_attrs)
add cache attribute to a memory node

Parameters

unsigned int nid Node identifier that has new cache attributes

struct node_cache_attrs * cache_attrs Attributes for the cache being added

void **unregister_node**(struct node * node)
unregister a node device

Parameters

struct node * node node going away

Description

Unregisters a node device **node**. All the devices on the node must be unregistered before calling this function.

int **register_memory_node_under_compute_node**(unsigned int mem_nid, unsigned int cpu_nid, unsigned access)
link memory node to its compute node for a given access class.

Parameters

unsigned int mem_nid Memory node number

unsigned int cpu_nid Cpu node number

unsigned access Access class to register

Description

For use with platforms that may have separate memory and compute nodes. This function will export node relationships linking which memory initiator nodes can access memory targets at a given ranked access class.

int **request_firmware**(const struct firmware ** firmware_p, const char * name, struct device * device)
send firmware request and wait for it

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded

firmware_p will be used to return a firmware image by the name of **name** for device **device**.

Should be called from user context where sleeping is allowed.

name will be used as $FIRMWARE in the uevent environment and should be distinctive enough not to be confused with any other firmware image for this or any other device.

Caller must hold the reference count of **device**.

The function can be called safely inside device's suspend and resume callback.

int **firmware_request_nowarn**(const struct firmware ** firmware, const char * name, struct device * device)
request for an optional fw module

Parameters

const struct firmware ** firmware pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded

Description

This function is similar in behaviour to `request_firmware()`, except it doesn't produce warning messages when the file is not found. The sysfs fallback mechanism is enabled if direct filesystem lookup fails, however, however failures to find the firmware file with it are still suppressed. It is therefore up to the driver to check for the return value of this call and to decide when to inform the users of errors.

int **request_firmware_direct**(const struct firmware ** firmware_p, const char * name, struct device * device)
load firmware directly without usermode helper

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded

Description

This function works pretty much like `request_firmware()`, but this doesn't fall back to usermode helper even if the firmware couldn't be loaded directly from fs. Hence it's useful for loading optional firmwares, which aren't always present, without extra long timeouts of udev.

int **firmware_request_platform**(const struct firmware ** firmware, const char * name, struct device * device)
request firmware with platform-fw fallback

Parameters

const struct firmware ** firmware pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded

Description

This function is similar in behaviour to request_firmware, except that if direct filesystem lookup fails, it will fallback to looking for a copy of the requested firmware embedded in the platform's main (e.g. UEFI) firmware.

int **firmware_request_cache**(struct device * device, const char * name)
cache firmware for suspend so resume can use it

Parameters

struct device * device device for which firmware should be cached for

const char * name name of firmware file

Description

There are some devices with an optimization that enables the device to not require loading firmware on system reboot. This optimization may still require the firmware present on resume from suspend. This routine can be used to ensure the firmware is present on resume from suspend in these situations. This helper is not compatible with drivers which use `request_firmware_into_buf()` or `request_firmware_nowait()` with no uevent set.

int **request_firmware_into_buf**(const struct firmware ** firmware_p, const char * name, struct device * device, void * buf, size_t size)
load firmware into a previously allocated buffer

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded and DMA region allocated

void * buf address of buffer to load firmware into

size_t size size of buffer

Description

This function works pretty much like `request_firmware()`, but it doesn't allocate a buffer to hold the firmware data. Instead, the firmware is loaded directly into the buffer pointed to by **buf** and the **firmware_p** data member is pointed at **buf**.

This function doesn't cache firmware either.

void **release_firmware**(const struct firmware * fw)
release the resource associated with a firmware image

Parameters

const struct firmware * fw firmware resource to release

int **request_firmware_nowait**(struct module * module, bool uevent, const char * name, struct device * device, gfp_t gfp, void * context, void (*cont)(const struct firmware *fw, void *context))
asynchronous version of request_firmware

Parameters

struct module * module module requesting the firmware

bool uevent sends uevent to copy the firmware image if this flag is non-zero else the firmware copy must be done manually.

const char * name name of firmware file

struct device * device device for which firmware is being loaded

gfp_t gfp allocation flags

void * context will be passed over to **cont**, and **fw** may be `NULL` if firmware request fails.

void (*)(const struct firmware *fw, void *context) cont function will be called asynchronously when the firmware request is over.

Caller must hold the reference count of **device**.

Asynchronous variant of request_firmware() for user contexts:

- sleep for as small periods as possible since it may increase kernel boot time of built-in device drivers requesting firmware in their ->probe() methods, if **gfp** is GFP_KERNEL.
- can't sleep at all if **gfp** is GFP_ATOMIC.

int **transport_class_register**(struct transport_class * tclass)
register an initial transport class

Parameters

struct transport_class * tclass a pointer to the transport class structure to be initialised

Description

The transport class contains an embedded class which is used to identify it. The caller should initialise this structure with zeros and then generic class must have been initialised with the actual transport class unique name. There's a macro DECLARE_TRANSPORT_CLASS() to do this (declared classes still must be registered).

Returns 0 on success or error on failure.

void **transport_class_unregister**(struct transport_class * tclass)
unregister a previously registered class

Parameters

struct transport_class * tclass The transport class to unregister

Description

Must be called prior to deallocating the memory for the transport class.

int **anon_transport_class_register**(struct anon_transport_class * atc)
register an anonymous class

Parameters

struct anon_transport_class * atc The anon transport class to register

Description

The anonymous transport class contains both a transport class and a container. The idea of an anonymous class is that it never actually has any device attributes associated with it (and thus saves on container storage). So it can only be used for triggering events. Use prezero and then use DECLARE_ANON_TRANSPORT_CLASS() to initialise the anon transport class storage.

void **anon_transport_class_unregister**(struct anon_transport_class * atc)
unregister an anon class

Parameters

struct anon_transport_class * atc Pointer to the anon transport class to unregister

Description

Must be called prior to deallocating the memory for the anon transport class.

void **transport_setup_device**(struct device * dev)
declare a new dev for transport class association but don't make it visible yet.

Parameters

struct device * dev the generic device representing the entity being added

Description

Usually, dev represents some component in the HBA system (either the HBA itself or a device remote across the HBA bus). This routine is simply a trigger point to see if any set of transport classes wishes to associate with the added device. This

allocates storage for the class device and initialises it, but does not yet add it to the system or add attributes to it (you do this with transport_add_device). If you have no need for a separate setup and add operations, use transport_register_device (see transport_class.h).

int **transport_add_device**(struct device * dev)
declare a new dev for transport class association

Parameters

struct device * dev the generic device representing the entity being added

Description

Usually, dev represents some component in the HBA system (either the HBA itself or a device remote across the HBA bus). This routine is simply a trigger point used to add the device to the system and register attributes for it.

void **transport_configure_device**(struct device * dev)
configure an already set up device

Parameters

struct device * dev generic device representing device to be configured

Description

The idea of configure is simply to provide a point within the setup process to allow the transport class to extract information from a device after it has been setup. This is used in SCSI because we have to have a setup device to begin using the HBA, but after we send the initial inquiry, we use configure to extract the device parameters. The device need not have been added to be configured.

void **transport_remove_device**(struct device * dev)
remove the visibility of a device

Parameters

struct device * dev generic device to remove

Description

This call removes the visibility of the device (to the user from sysfs), but does not destroy it. To eliminate a device entirely you must also call transport_destroy_device. If you don't need to do remove and destroy as separate operations, use transport_unregister_device() (see transport_class.h) which will perform both calls for you.

void **transport_destroy_device**(struct device * dev)
destroy a removed device

Parameters

struct device * dev device to eliminate from the transport class.

Description

This call triggers the elimination of storage associated with the transport classdev. Note: all it really does is relinquish a reference to the classdev. The memory will not be freed until the last reference goes to zero. Note also that the classdev

retains a reference count on dev, so dev too will remain for as long as the transport class device remains around.

int **device_bind_driver**(struct device * dev)
bind a driver to one device.

Parameters

struct device * dev device.

Description

Allow manual attachment of a driver to a device. Caller must have already set **dev->driver**.

Note that this does not modify the bus reference count nor take the bus's rwsem. Please verify those are accounted for before calling this. (It is ok to call with no other effort from a driver's probe() method.)

This function must be called with the device lock held.

void **wait_for_device_probe**(void)

Parameters

void no arguments

Description

Wait for device probing to be completed.

int **device_attach**(struct device * dev)
try to attach device to a driver.

Parameters

struct device * dev device.

Description

Walk the list of drivers that the bus has and call driver_probe_device() for each pair. If a compatible pair is found, break out and return.

Returns 1 if the device was bound to a driver; 0 if no matching driver was found; -ENODEV if the device is not registered.

When called for a USB interface, **dev->parent** lock must be held.

int **driver_attach**(struct device_driver * drv)
try to bind driver to devices.

Parameters

struct device_driver * drv driver.

Description

Walk the list of devices that the bus has on it and try to match the driver with each one. If driver_probe_device() returns 0 and the **dev->driver** is set, we've found a compatible pair.

void **device_release_driver**(struct device * dev)
manually detach device from driver.

Parameters

struct device * dev device.

Description

Manually detach device from driver. When called for a USB interface, **dev->parent** lock must be held.

If this function is to be called with **dev->parent** lock held, ensure that the device's consumers are unbound in advance or that their locks can be acquired under the **dev->parent** lock.

struct platform_device * **platform_device_register_resndata**(struct device * parent, const char * name, int id, const struct resource * res, unsigned int num, const void * data, size_t size)

add a platform-level device with resources and platform-specific data

Parameters

struct device * parent parent device for the device we're adding

const char * name base name of the device we're adding

int id instance id

const struct resource * res set of resources that needs to be allocated for the device

unsigned int num number of resources

const void * data platform specific data for this platform device

size_t size size of platform specific data

Description

Returns `struct platform_device` pointer on success, or ERR_PTR() on error.

struct platform_device * **platform_device_register_simple**(const char * name, int id, const struct resource * res, unsigned int num)

add a platform-level device and its resources

Parameters

const char * name base name of the device we're adding

int id instance id

const struct resource * res set of resources that needs to be allocated for the device

unsigned int num number of resources

Description

This function creates a simple platform device that requires minimal resource and memory management. Canned release function freeing memory allocated for the device allows drivers using such devices to be unloaded without waiting for the last reference to the device to be dropped.

This interface is primarily intended for use with legacy drivers which probe hardware directly. Because such drivers create sysfs device nodes themselves, rather than letting system infrastructure handle such device enumeration tasks, they don't fully conform to the Linux driver model. In particular, when such drivers are built as modules, they can't be "hotplugged".

Returns `struct platform_device` pointer on success, or ERR_PTR() on error.

struct platform_device * **platform_device_register_data**(struct device * parent, const char * name, int id, const void * data, size_t size)

add a platform-level device with platform-specific data

Parameters

struct device * parent parent device for the device we're adding

const char * name base name of the device we're adding

int id instance id

const void * data platform specific data for this platform device

size_t size size of platform specific data

Description

This function creates a simple platform device that requires minimal resource and memory management. Canned release function freeing memory allocated for the device allows drivers using such devices to be unloaded without waiting for the last reference to the device to be dropped.

Returns `struct platform_device` pointer on success, or ERR_PTR() on error.

struct resource * **platform_get_resource**(struct platform_device * dev, unsigned int type, unsigned int num)

get a resource for a device

Parameters

struct platform_device * dev platform device

unsigned int type resource type

unsigned int num resource index

void __iomem * **devm_platform_get_and_ioremap_resource**(struct platform_device * pdev, unsigned int index, struct resource ** res)

call devm_ioremap_resource() for a platform device and get resource

Parameters

struct platform_device * pdev platform device to use both for memory resource lookup as well as resource management

unsigned int index resource index

struct resource ** res optional output parameter to store a pointer to the obtained resource.

void __iomem * **devm_platform_ioremap_resource**(struct platform_device * pdev, unsigned int index)

call devm_ioremap_resource() for a platform device

Parameters

struct platform_device * pdev platform device to use both for memory resource lookup as well as resource management

unsigned int index resource index

void __iomem * **devm_platform_ioremap_resource_byname**(struct platform_device * pdev, const char * name)

call devm_ioremap_resource for a platform device, retrieve the resource by name

Parameters

struct platform_device * pdev platform device to use both for memory resource lookup as well as resource management

const char * name name of the resource

int **platform_get_irq_optional**(struct platform_device * dev, unsigned int num)

get an optional IRQ for a device

Parameters

struct platform_device * dev platform device

unsigned int num IRQ number index

Description

Gets an IRQ for a platform device. Device drivers should check the return value for errors so as to not pass a negative integer value to the request_irq() APIs. This is the same as `platform_get_irq()`, except that it does not print an error message if an IRQ can not be obtained.

For example:

```
int irq = platform_get_irq_optional(pdev, 0);
if (irq < 0)
        return irq;
```

Return

non-zero IRQ number on success, negative error number on failure.

int **platform_get_irq**(struct platform_device * dev, unsigned int num)
get an IRQ for a device

Parameters

struct platform_device * dev platform device

unsigned int num IRQ number index

Description

Gets an IRQ for a platform device and prints an error message if finding the IRQ fails. Device drivers should check the return value for errors so as to not pass a negative integer value to the request_irq() APIs.

For example:

```
int irq = platform_get_irq(pdev, 0);
if (irq < 0)
        return irq;
```

Return

non-zero IRQ number on success, negative error number on failure.

int **platform_irq_count**(struct platform_device * dev)
Count the number of IRQs a platform device uses

Parameters

struct platform_device * dev platform device

Return

Number of IRQs a platform device uses or EPROBE_DEFER

struct resource * **platform_get_resource_byname**(struct platform_device * dev, unsigned int type, const char * name)
get a resource for a device by name

Parameters

struct platform_device * dev platform device

unsigned int type resource type

const char * name resource name

int **platform_get_irq_byname**(struct platform_device * dev, const char * name)
get an IRQ for a device by name

Parameters

struct platform_device * dev platform device

const char * name IRQ name

Description

Get an IRQ like `platform_get_irq()`, but then by name rather then by index.

Return

non-zero IRQ number on success, negative error number on failure.

int **platform_get_irq_byname_optional**(struct platform_device * dev, const char * name)
get an optional IRQ for a device by name

Parameters

struct platform_device * dev platform device

const char * name IRQ name

Description

Get an optional IRQ by name like `platform_get_irq_byname()`. Except that it does not print an error message if an IRQ can not be obtained.

Return

non-zero IRQ number on success, negative error number on failure.

int **platform_add_devices**(struct platform_device ** devs, int num)
add a numbers of platform devices

Parameters

struct platform_device ** devs array of platform devices to add

int num number of platform devices in array

void **platform_device_put**(struct platform_device * pdev)
destroy a platform device

Parameters

struct platform_device * pdev platform device to free

Description

Free all memory associated with a platform device. This function must _only_ be externally called in error cases. All other usage is a bug.

struct platform_device * **platform_device_alloc**(const char * name, int id)
create a platform device

Parameters

const char * name base name of the device we’re adding

int id instance id

Description

Create a platform device object which can have other objects attached to it, and which will have attached objects freed when it is released.

int **platform_device_add_resources**(struct platform_device * pdev, const struct resource * res, unsigned int num)
add resources to a platform device

Parameters

struct platform_device * pdev platform device allocated by platform_device_alloc to add resources to

const struct resource * res set of resources that needs to be allocated for the device

unsigned int num number of resources

Description

Add a copy of the resources to the platform device. The memory associated with the resources will be freed when the platform device is released.

int **platform_device_add_data**(struct platform_device * pdev, const void * data, size_t size)
add platform-specific data to a platform device

Parameters

struct platform_device * pdev platform device allocated by platform_device_alloc to add resources to

const void * data platform specific data for this platform device

size_t size size of platform specific data

Description

Add a copy of platform specific data to the platform device's platform_data pointer. The memory associated with the platform data will be freed when the platform device is released.

int **platform_device_add_properties**(struct platform_device * pdev, const struct property_entry * properties)
add built-in properties to a platform device

Parameters

struct platform_device * pdev platform device to add properties to

const struct property_entry * properties null terminated array of properties to add

Description

The function will take deep copy of **properties** and attach the copy to the platform device. The memory associated with properties will be freed when the platform device is released.

int **platform_device_add**(struct platform_device * pdev)
add a platform device to device hierarchy

Parameters

struct platform_device * pdev platform device we’re adding

Description

This is part 2 of `platform_device_register()`, though may be called separately _iff_ pdev was allocated by `platform_device_alloc()`.

void **platform_device_del**(struct platform_device * pdev)
remove a platform-level device

Parameters

struct platform_device * pdev platform device we’re removing

Description

Note that this function will also release all memory- and port-based resources owned by the device (**dev->resource**). This function must _only_ be externally called in error cases. All other usage is a bug.

int **platform_device_register**(struct platform_device * pdev)
add a platform-level device

Parameters

struct platform_device * pdev platform device we’re adding

void **platform_device_unregister**(struct platform_device * pdev)
unregister a platform-level device

Parameters

struct platform_device * pdev platform device we’re unregistering

Description

Unregistration is done in 2 steps. First we release all resources and remove it from the subsystem, then we drop reference count by calling `platform_device_put()`.

struct platform_device * **platform_device_register_full**(const struct platform_device_info * pdevinfo)
add a platform-level device with resources and platform-specific data

Parameters

const struct platform_device_info * pdevinfo data used to create device

Description

Returns `struct platform_device` pointer on success, or ERR_PTR() on error.

int **__platform_driver_register**(struct platform_driver * drv, struct module * owner)
register a driver for platform-level devices

Parameters

struct platform_driver * drv platform driver structure

struct module * owner owning module/driver

void **platform_driver_unregister**(struct platform_driver * drv)
unregister a driver for platform-level devices

Parameters

struct platform_driver * drv platform driver structure

int **__platform_driver_probe**(struct platform_driver * drv, int (*probe)(struct platform_device *), struct module * module)
register driver for non-hotpluggable device

Parameters

struct platform_driver * drv platform driver structure

int (*)(struct platform_device *) probe the driver probe routine, probably from an __init section

struct module * module module which will be the owner of the driver

Description

Use this instead of platform_driver_register() when you know the device is not hotpluggable and has already been registered, and you want to remove its run-once probe() infrastructure from memory after the driver has bound to the device.

One typical use for this would be with drivers for controllers integrated into system-on-chip processors, where the controller devices have been configured as part of board setup.

Note that this is incompatible with deferred probing.

Returns zero if the driver registered and bound to a device, else returns a negative error code and with the driver not registered.

struct platform_device * **__platform_create_bundle**(struct platform_driver * driver, int (*probe)(struct platform_device *), struct resource * res, unsigned int n_res, const void * data, size_t size, struct module * module)
register driver and create corresponding device

Parameters

struct platform_driver * driver platform driver structure

int (*)(struct platform_device *) probe the driver probe routine, probably from an __init section

struct resource * res set of resources that needs to be allocated for the device

unsigned int n_res number of resources

const void * data platform specific data for this platform device

size_t size size of platform specific data

struct module * module module which will be the owner of the driver

Description

Use this in legacy-style modules that probe hardware directly and register a single platform device and corresponding platform driver.

Returns `struct platform_device` pointer on success, or ERR_PTR() on error.

int **__platform_register_drivers**(struct platform_driver *const * drivers, unsigned int count, struct module * owner)

register an array of platform drivers

Parameters

struct platform_driver *const * drivers an array of drivers to register

unsigned int count the number of drivers to register

struct module * owner module owning the drivers

Description

Registers platform drivers specified by an array. On failure to register a driver, all previously registered drivers will be unregistered. Callers of this API should use `platform_unregister_drivers()` to unregister drivers in the reverse order.

Return

0 on success or a negative error code on failure.

void **platform_unregister_drivers**(struct platform_driver *const * drivers, unsigned int count)

unregister an array of platform drivers

Parameters

struct platform_driver *const * drivers an array of drivers to unregister

unsigned int count the number of drivers to unregister

Description

Unregisters platform drivers specified by an array. This is typically used to complement an earlier call to platform_register_drivers(). Drivers are unregistered in the reverse order in which they were registered.

struct device * **platform_find_device_by_driver**(struct device * start, const struct device_driver * drv)

Find a platform device with a given driver.

Parameters

struct device * start The device to start the search from.

const struct device_driver * drv The device driver to look for.

int **bus_for_each_dev**(struct bus_type * bus, struct device * start, void * data, int (*fn)(struct device *, void *))
device iterator.

Parameters

struct bus_type * bus bus type.

struct device * start device to start iterating from.

void * data data for the callback.

int (*)(struct device *, void *) fn function to be called for each device.

Description

Iterate over **bus**'s list of devices, and call **fn** for each, passing it **data**. If **start** is not NULL, we use that device to begin iterating from.

We check the return of **fn** each time. If it returns anything other than 0, we break out and return that value.

NOTE

The device that returns a non-zero value is not retained in any way, nor is its refcount incremented. If the caller needs to retain this data, it should do so, and increment the reference count in the supplied callback.

struct device * **bus_find_device**(struct bus_type * bus, struct device * start, const void * data, int (*match)(struct device *dev, const void *data))
device iterator for locating a particular device.

Parameters

struct bus_type * bus bus type

struct device * start Device to begin with

const void * data Data to pass to match function

int (*)(struct device *dev, const void *data) match Callback function to check device

Description

This is similar to the `bus_for_each_dev()` function above, but it returns a reference to a device that is 'found' for later use, as determined by the **match** callback.

The callback should return 0 if the device doesn't match and non-zero if it does. If the callback returns non-zero, this function will return to the caller and not iterate over any more devices.

struct device * **subsys_find_device_by_id**(struct bus_type * subsys, unsigned int id, struct device * hint)
find a device with a specific enumeration number

Parameters

struct bus_type * subsys subsystem

unsigned int id index 'id' in struct device

struct device * hint device to check first

Description

Check the hint's next object and if it is a match return it directly, otherwise, fall back to a full list search. Either way a reference for the returned object is taken.

int **bus_for_each_drv**(struct bus_type * bus, struct device_driver * start, void * data, int (*fn)(struct device_driver *, void *))
driver iterator

Parameters

struct bus_type * bus bus we're dealing with.

struct device_driver * start driver to start iterating on.

void * data data to pass to the callback.

int (*)(struct device_driver *, void *) fn function to call for each driver.

Description

This is nearly identical to the device iterator above. We iterate over each driver that belongs to **bus**, and call **fn** for each. If **fn** returns anything but 0, we break out and return it. If **start** is not NULL, we use it as the head of the list.

NOTE

we don't return the driver that returns a non-zero value, nor do we leave the reference count incremented for that driver. If the caller needs to know that info, it must set it in the callback. It must also be sure to increment the refcount so it doesn't disappear before returning to the caller.

int **bus_rescan_devices**(struct bus_type * bus)
rescan devices on the bus for possible drivers

Parameters

struct bus_type * bus the bus to scan.

Description

This function will look for devices on the bus with no driver attached and rescan it against existing drivers to see if it matches any by calling `device_attach()` for the unbound devices.

int **device_reprobe**(struct device * dev)
remove driver for a device and probe for a new driver

Parameters

struct device * dev the device to reprobe

Description

This function detaches the attached driver (if any) for the given device and restarts the driver probing process. It is intended to use if probing criteria changed during a devices lifetime and driver attachment should change accordingly.

int **bus_register**(struct bus_type * bus)
register a driver-core subsystem

Parameters

struct bus_type * bus bus to register

Description

Once we have that, we register the bus with the kobject infrastructure, then register the children subsystems it has: the devices and drivers that belong to the subsystem.

void **bus_unregister**(struct bus_type * bus)
remove a bus from the system

Parameters

struct bus_type * bus bus.

Description

Unregister the child subsystems and the bus itself. Finally, we call bus_put() to release the refcount

void **subsys_dev_iter_init**(struct subsys_dev_iter * iter, struct bus_type * subsys, struct device * start, const struct device_type * type)
initialize subsys device iterator

Parameters

struct subsys_dev_iter * iter subsys iterator to initialize

struct bus_type * subsys the subsys we wanna iterate over

struct device * start the device to start iterating from, if any

const struct device_type * type device_type of the devices to iterate over, NULL for all

Description

Initialize subsys iterator **iter** such that it iterates over devices of **subsys**. If **start** is set, the list iteration will start there, otherwise if it is NULL, the iteration starts at the beginning of the list.

struct device * **subsys_dev_iter_next**(struct subsys_dev_iter * iter)
iterate to the next device

Parameters

struct subsys_dev_iter * iter subsys iterator to proceed

Description

Proceed **iter** to the next device and return it. Returns NULL if iteration is complete.

The returned device is referenced and won't be released till iterator is proceed to the next device or exited. The caller is free to do whatever it wants to do with the device including calling back into subsys code.

void **subsys_dev_iter_exit**(struct subsys_dev_iter * iter)
finish iteration

Parameters

struct subsys_dev_iter * iter subsys iterator to finish

Description

Finish an iteration. Always call this function after iteration is complete whether the iteration ran till the end or not.

int **subsys_system_register**(struct bus_type * subsys, const struct attribute_group ** groups)
register a subsystem at /sys/devices/system/

Parameters

struct bus_type * subsys system subsystem

const struct attribute_group ** groups default attributes for the root device

Description

All 'system' subsystems have a /sys/devices/system/<name> root device with the name of the subsystem. The root device can carry subsystem- wide attributes. All registered devices are below this single root device and are named after the subsystem with a simple enumeration number appended. The registered devices are not explicitly named; only 'id' in the device needs to be set.

Do not use this interface for anything new, it exists for compatibility with bad ideas only. New subsystems should use plain subsystems; and add the subsystem-wide attributes should be added to the subsystem directory itself and not some create fake root-device placed in /sys/devices/system/<name>.

int **subsys_virtual_register**(struct bus_type * subsys, const struct attribute_group ** groups)
register a subsystem at /sys/devices/virtual/

Parameters

struct bus_type * subsys virtual subsystem

const struct attribute_group ** groups default attributes for the root device

Description

All 'virtual' subsystems have a /sys/devices/system/<name> root device with the name of the subystem. The root device can carry subsystem-wide attributes. All registered devices are below this single root device. There's no restriction on device naming. This is for kernel software constructs which need sysfs interface.

3.3 Device Drivers DMA Management

void **dmam_free_coherent**(struct device * dev, size_t size, void * vaddr, dma_addr_t dma_handle)
Managed dma_free_coherent()

Parameters

struct device * dev Device to free coherent memory for

size_t size Size of allocation

void * vaddr Virtual address of the memory to free

dma_addr_t dma_handle DMA handle of the memory to free

Description

Managed dma_free_coherent().

void * **dmam_alloc_attrs**(struct device * dev, size_t size, dma_addr_t * dma_handle, gfp_t gfp, unsigned long attrs)
Managed dma_alloc_attrs()

Parameters

struct device * dev Device to allocate non_coherent memory for

size_t size Size of allocation

dma_addr_t * dma_handle Out argument for allocated DMA handle

gfp_t gfp Allocation flags

unsigned long attrs Flags in the DMA_ATTR_* namespace.

Description

Managed dma_alloc_attrs(). Memory allocated using this function will be automatically released on driver detach.

Return

Pointer to allocated memory on success, NULL on failure.

bool **dma_can_mmap**(struct device * dev)
check if a given device supports dma_mmap_*

Parameters

struct device * dev device to check

Description

Returns `true` if **dev** supports dma_mmap_coherent() and `dma_mmap_attrs()` to map DMA allocations to userspace.

int **dma_mmap_attrs**(struct device * dev, struct vm_area_struct * vma, void * cpu_addr, dma_addr_t dma_addr, size_t size, unsigned long attrs)
map a coherent DMA allocation into user space

Parameters

struct device * dev valid struct device pointer, or NULL for ISA and EISA-like devices

struct vm_area_struct * vma vm_area_struct describing requested user mapping

void * cpu_addr kernel CPU-view address returned from dma_alloc_attrs

dma_addr_t dma_addr device-view address returned from dma_alloc_attrs

size_t size size of memory originally requested in dma_alloc_attrs

unsigned long attrs attributes of mapping properties requested in dma_alloc_attrs

Description

Map a coherent DMA buffer previously allocated by dma_alloc_attrs into user space. The coherent DMA buffer must not be freed by the driver until the user space mapping has been released.

3.4 Device drivers PnP support

int **pnp_register_protocol**(struct pnp_protocol * protocol)
adds a pnp protocol to the pnp layer

Parameters

struct pnp_protocol * protocol pointer to the corresponding pnp_protocol structure

Ex protocols: ISAPNP, PNPBIOS, etc

void **pnp_unregister_protocol**(struct pnp_protocol * protocol)
removes a pnp protocol from the pnp layer

Parameters

struct pnp_protocol * protocol pointer to the corresponding pnp_protocol structure

struct pnp_dev * **pnp_request_card_device**(struct pnp_card_link * clink, const char * id, struct pnp_dev * from)
Searches for a PnP device under the specified card

Parameters

struct pnp_card_link * clink pointer to the card link, cannot be NULL

const char * id pointer to a PnP ID structure that explains the rules for finding the device

struct pnp_dev * from Starting place to search from. If NULL it will start from the beginning.

void **pnp_release_card_device**(struct pnp_dev * dev)
call this when the driver no longer needs the device

Parameters

struct pnp_dev * dev pointer to the PnP device structure

int **pnp_register_card_driver**(struct pnp_card_driver * drv)
registers a PnP card driver with the PnP Layer

Parameters

struct pnp_card_driver * drv pointer to the driver to register

void **pnp_unregister_card_driver**(struct pnp_card_driver * drv)
unregisters a PnP card driver from the PnP Layer

Parameters

struct pnp_card_driver * drv pointer to the driver to unregister

struct pnp_id * **pnp_add_id**(struct pnp_dev * dev, const char * id)
adds an EISA id to the specified device

Parameters

struct pnp_dev * dev pointer to the desired device

const char * id pointer to an EISA id string

int **pnp_start_dev**(struct pnp_dev * dev)
low-level start of the PnP device

Parameters

struct pnp_dev * dev pointer to the desired device

Description

assumes that resources have already been allocated

int **pnp_stop_dev**(struct pnp_dev * dev)
low-level disable of the PnP device

Parameters

struct pnp_dev * dev pointer to the desired device

Description

does not free resources

int **pnp_activate_dev**(struct pnp_dev * dev)
activates a PnP device for use

Parameters

struct pnp_dev * dev pointer to the desired device

Description

does not validate or set resources so be careful.

int **pnp_disable_dev**(struct pnp_dev * dev)
disables device

Parameters

struct pnp_dev * dev pointer to the desired device

Description

inform the correct pnp protocol so that resources can be used by other devices

int **pnp_is_active**(struct pnp_dev * dev)
Determines if a device is active based on its current resources

Parameters

struct pnp_dev * dev pointer to the desired PnP device

3.5 Userspace IO devices

void **uio_event_notify**(struct uio_info * info)
trigger an interrupt event

Parameters

struct uio_info * info UIO device capabilities

int **__uio_register_device**(struct module * owner, struct device * parent, struct uio_info * info)
register a new userspace IO device

Parameters

struct module * owner module that creates the new device

struct device * parent parent device

struct uio_info * info UIO device capabilities

Description

returns zero on success or a negative error code.

int **__devm_uio_register_device**(struct module * owner, struct device * parent, struct uio_info * info)
Resource managed uio_register_device()

Parameters

struct module * owner module that creates the new device

struct device * parent parent device

struct uio_info * info UIO device capabilities

Description

returns zero on success or a negative error code.

void **uio_unregister_device**(struct uio_info * info)
unregister a industrial IO device

Parameters

struct uio_info * info UIO device capabilities

struct **uio_mem**
description of a UIO memory region

Definition

```
struct uio_mem {
  const char              *name;
  phys_addr_t addr;
  unsigned long           offs;
  resource_size_t size;
  int memtype;
  void __iomem            *internal_addr;
  struct uio_map          *map;
};
```

Members

name name of the memory region for identification

addr address of the device’s memory rounded to page size (phys_addr is used since addr can be logical, virtual, or physical & phys_addr_t should always be large enough to handle any of the address types)

offs offset of device memory within the page

size size of IO (multiple of page size)

memtype type of memory addr points to

internal_addr ioremap-ped version of addr, for driver internal use

map for use by the UIO core only.

struct **uio_port**
description of a UIO port region

Definition

```
struct uio_port {
  const char              *name;
  unsigned long           start;
  unsigned long           size;
  int porttype;
  struct uio_portio       *portio;
};
```

Members

name name of the port region for identification

start start of port region

size size of port region

porttype type of port (see UIO_PORT_* below)

portio for use by the UIO core only.

struct **uio_info**
UIO device capabilities

Definition

```
struct uio_info {
  struct uio_device       *uio_dev;
```

(continues on next page)

(continued from previous page)

```
  const char              *name;
  const char              *version;
  struct uio_mem          mem[MAX_UIO_MAPS];
  struct uio_port         port[MAX_UIO_PORT_REGIONS];
  long irq;
  unsigned long           irq_flags;
  void *priv;
  irqreturn_t (*handler)(int irq, struct uio_info *dev_info);
  int (*mmap)(struct uio_info *info, struct vm_area_struct *vma);
  int (*open)(struct uio_info *info, struct inode *inode);
  int (*release)(struct uio_info *info, struct inode *inode);
  int (*irqcontrol)(struct uio_info *info, s32 irq_on);
};
```

Members

uio_dev the UIO device this info belongs to

name device name

version device driver version

mem list of mappable memory regions, size==0 for end of list

port list of port regions, size==0 for end of list

irq interrupt number or UIO_IRQ_CUSTOM

irq_flags flags for request_irq()

priv optional private data

handler the device’s irq handler

mmap mmap operation for this uio device

open open operation for this uio device

release release operation for this uio device

irqcontrol disable/enable irqs when 0/1 is written to /dev/uioX

CHAPTER

FOUR

IOCTL BASED INTERFACES

ioctl() is the most common way for applications to interface with device drivers. It is flexible and easily extended by adding new commands and can be passed through character devices, block devices as well as sockets and other special file descriptors.

However, it is also very easy to get ioctl command definitions wrong, and hard to fix them later without breaking existing applications, so this documentation tries to help developers get it right.

4.1 Command number definitions

The command number, or request number, is the second argument passed to the ioctl system call. While this can be any 32-bit number that uniquely identifies an action for a particular driver, there are a number of conventions around defining them.

`include/uapi/asm-generic/ioctl.h` provides four macros for defining ioctl commands that follow modern conventions: `_IO`, `_IOR`, `_IOW`, and `_IOWR`. These should be used for all new commands, with the correct parameters:

_IO/_IOR/_IOW/_IOWR The macro name specifies how the argument will be used. It may be a pointer to data to be passed into the kernel (_IOW), out of the kernel (_IOR), or both (_IOWR). _IO can indicate either commands with no argument or those passing an integer value instead of a pointer. It is recommended to only use _IO for commands without arguments, and use pointers for passing data.

type An 8-bit number, often a character literal, specific to a subsystem or driver, and listed in ../userspace-api/ioctl/ioctl-number

nr An 8-bit number identifying the specific command, unique for a give value of 'type'

data_type The name of the data type pointed to by the argument, the command number encodes the `sizeof(data_type)` value in a 13-bit or 14-bit integer, leading to a limit of 8191 bytes for the maximum size of the argument. Note: do not pass sizeof(data_type) type into _IOR/_IOW/IOWR, as that will lead to encoding sizeof(sizeof(data_type)), i.e. sizeof(size_t). _IO does not have a data_type parameter.

4.2 Interface versions

Some subsystems use version numbers in data structures to overload commands with different interpretations of the argument.

This is generally a bad idea, since changes to existing commands tend to break existing applications.

A better approach is to add a new ioctl command with a new number. The old command still needs to be implemented in the kernel for compatibility, but this can be a wrapper around the new implementation.

4.3 Return code

ioctl commands can return negative error codes as documented in errno(3); these get turned into errno values in user space. On success, the return code should be zero. It is also possible but not recommended to return a positive 'long' value.

When the ioctl callback is called with an unknown command number, the handler returns either -ENOTTY or -ENOIOCTLCMD, which also results in -ENOTTY being returned from the system call. Some subsystems return -ENOSYS or -EINVAL here for historic reasons, but this is wrong.

Prior to Linux 5.5, compat_ioctl handlers were required to return -ENOIOCTLCMD in order to use the fallback conversion into native commands. As all subsystems are now responsible for handling compat mode themselves, this is no longer needed, but it may be important to consider when backporting bug fixes to older kernels.

4.4 Timestamps

Traditionally, timestamps and timeout values are passed as `struct timespec` or `struct timeval`, but these are problematic because of incompatible definitions of these structures in user space after the move to 64-bit time_t.

The `struct __kernel_timespec` type can be used instead to be embedded in other data structures when separate second/nanosecond values are desired, or passed to user space directly. This is still not ideal though, as the structure matches neither the kernel's timespec64 nor the user space timespec exactly. The get_timespec64() and put_timespec64() helper functions can be used to ensure that the layout remains compatible with user space and the padding is treated correctly.

As it is cheap to convert seconds to nanoseconds, but the opposite requires an expensive 64-bit division, a simple __u64 nanosecond value can be simpler and more efficient.

Timeout values and timestamps should ideally use CLOCK_MONOTONIC time, as returned by ktime_get_ns() or ktime_get_ts64(). Unlike CLOCK_REALTIME, this makes the timestamps immune from jumping backwards or forwards due to leap second adjustments and clock_settime() calls.

ktime_get_real_ns() can be used for CLOCK_REALTIME timestamps that need to be persistent across a reboot or between multiple machines.

4.5 32-bit compat mode

In order to support 32-bit user space running on a 64-bit machine, each subsystem or driver that implements an ioctl callback handler must also implement the corresponding compat_ioctl handler.

As long as all the rules for data structures are followed, this is as easy as setting the .compat_ioctl pointer to a helper function such as compat_ptr_ioctl() or blkdev_compat_ptr_ioctl().

4.5.1 compat_ptr()

On the s390 architecture, 31-bit user space has ambiguous representations for data pointers, with the upper bit being ignored. When running such a process in compat mode, the compat_ptr() helper must be used to clear the upper bit of a compat_uptr_t and turn it into a valid 64-bit pointer. On other architectures, this macro only performs a cast to a `void __user *` pointer.

In an compat_ioctl() callback, the last argument is an unsigned long, which can be interpreted as either a pointer or a scalar depending on the command. If it is a scalar, then compat_ptr() must not be used, to ensure that the 64-bit kernel behaves the same way as a 32-bit kernel for arguments with the upper bit set.

The compat_ptr_ioctl() helper can be used in place of a custom compat_ioctl file operation for drivers that only take arguments that are pointers to compatible data structures.

4.5.2 Structure layout

Compatible data structures have the same layout on all architectures, avoiding all problematic members:

- `long` and `unsigned long` are the size of a register, so they can be either 32-bit or 64-bit wide and cannot be used in portable data structures. Fixed-length replacements are `__s32`, `__u32`, `__s64` and `__u64`.
- Pointers have the same problem, in addition to requiring the use of compat_ptr(). The best workaround is to use `__u64` in place of pointers, which requires a cast to `uintptr_t` in user space, and the use of u64_to_user_ptr() in the kernel to convert it back into a user pointer.
- On the x86-32 (i386) architecture, the alignment of 64-bit variables is only 32-bit, but they are naturally aligned on most other architectures including x86-64. This means a structure like:

```
struct foo {
    __u32 a;
    __u64 b;
```

(continues on next page)

(continued from previous page)

```
    __u32 c;
};
```

has four bytes of padding between a and b on x86-64, plus another four bytes of padding at the end, but no padding on i386, and it needs a compat_ioctl conversion handler to translate between the two formats.

To avoid this problem, all structures should have their members naturally aligned, or explicit reserved fields added in place of the implicit padding. The `pahole` tool can be used for checking the alignment.

- On ARM OABI user space, structures are padded to multiples of 32-bit, making some structs incompatible with modern EABI kernels if they do not end on a 32-bit boundary.
- On the m68k architecture, struct members are not guaranteed to have an alignment greater than 16-bit, which is a problem when relying on implicit padding.
- Bitfields and enums generally work as one would expect them to, but some properties of them are implementation-defined, so it is better to avoid them completely in ioctl interfaces.
- `char` members can be either signed or unsigned, depending on the architecture, so the __u8 and __s8 types should be used for 8-bit integer values, though char arrays are clearer for fixed-length strings.

4.6 Information leaks

Uninitialized data must not be copied back to user space, as this can cause an information leak, which can be used to defeat kernel address space layout randomization (KASLR), helping in an attack.

For this reason (and for compat support) it is best to avoid any implicit padding in data structures. Where there is implicit padding in an existing structure, kernel drivers must be careful to fully initialize an instance of the structure before copying it to user space. This is usually done by calling memset() before assigning to individual members.

4.7 Subsystem abstractions

While some device drivers implement their own ioctl function, most subsystems implement the same command for multiple drivers. Ideally the subsystem has an .ioctl() handler that copies the arguments from and to user space, passing them into subsystem specific callback functions through normal kernel pointers.

This helps in various ways:

- Applications written for one driver are more likely to work for another one in the same subsystem if there are no subtle differences in the user space ABI.

- The complexity of user space access and data structure layout is done in one place, reducing the potential for implementation bugs.
- It is more likely to be reviewed by experienced developers that can spot problems in the interface when the ioctl is shared between multiple drivers than when it is only used in a single driver.

4.8 Alternatives to ioctl

There are many cases in which ioctl is not the best solution for a problem. Alternatives include:

- System calls are a better choice for a system-wide feature that is not tied to a physical device or constrained by the file system permissions of a character device node
- netlink is the preferred way of configuring any network related objects through sockets.
- debugfs is used for ad-hoc interfaces for debugging functionality that does not need to be exposed as a stable interface to applications.
- sysfs is a good way to expose the state of an in-kernel object that is not tied to a file descriptor.
- configfs can be used for more complex configuration than sysfs
- A custom file system can provide extra flexibility with a simple user interface but adds a lot of complexity to the implementation.

CHAPTER

FIVE

EARLY USERSPACE

5.1 Early userspace support

Last update: 2004-12-20 tlh

“Early userspace” is a set of libraries and programs that provide various pieces of functionality that are important enough to be available while a Linux kernel is coming up, but that don’t need to be run inside the kernel itself.

It consists of several major infrastructure components:

- gen_init_cpio, a program that builds a cpio-format archive containing a root filesystem image. This archive is compressed, and the compressed image is linked into the kernel image.
- initramfs, a chunk of code that unpacks the compressed cpio image midway through the kernel boot process.
- klibc, a userspace C library, currently packaged separately, that is optimized for correctness and small size.

The cpio file format used by initramfs is the “newc” (aka “cpio -H newc”) format, and is documented in the file “buffer-format.txt”. There are two ways to add an early userspace image: specify an existing cpio archive to be used as the image or have the kernel build process build the image from specifications.

5.1.1 CPIO ARCHIVE method

You can create a cpio archive that contains the early userspace image. Your cpio archive should be specified in CONFIG_INITRAMFS_SOURCE and it will be used directly. Only a single cpio file may be specified in CONFIG_INITRAMFS_SOURCE and directory and file names are not allowed in combination with a cpio archive.

5.1.2 IMAGE BUILDING method

The kernel build process can also build an early userspace image from source parts rather than supplying a cpio archive. This method provides a way to create images with root-owned files even though the image was built by an unprivileged user.

The image is specified as one or more sources in CONFIG_INITRAMFS_SOURCE. Sources can be either directories or files - cpio archives are not allowed when building from sources.

A source directory will have it and all of its contents packaged. The specified directory name will be mapped to '/'. When packaging a directory, limited user and group ID translation can be performed. INITRAMFS_ROOT_UID can be set to a user ID that needs to be mapped to user root (0). INITRAMFS_ROOT_GID can be set to a group ID that needs to be mapped to group root (0).

A source file must be directives in the format required by the usr/gen_init_cpio utility (run 'usr/gen_init_cpio -h' to get the file format). The directives in the file will be passed directly to usr/gen_init_cpio.

When a combination of directories and files are specified then the initramfs image will be an aggregate of all of them. In this way a user can create a 'root-image' directory and install all files into it. Because device-special files cannot be created by a unprivileged user, special files can be listed in a 'root-files' file. Both 'root-image' and 'root-files' can be listed in CONFIG_INITRAMFS_SOURCE and a complete early userspace image can be built by an unprivileged user.

As a technical note, when directories and files are specified, the entire CONFIG_INITRAMFS_SOURCE is passed to usr/gen_initramfs_list.sh. This means that CONFIG_INITRAMFS_SOURCE can really be interpreted as any legal argument to gen_initramfs_list.sh. If a directory is specified as an argument then the contents are scanned, uid/gid translation is performed, and usr/gen_init_cpio file directives are output. If a directory is specified as an argument to usr/gen_initramfs_list.sh then the contents of the file are simply copied to the output. All of the output directives from directory scanning and file contents copying are processed by usr/gen_init_cpio.

See also 'usr/gen_initramfs_list.sh -h'.

Where's this all leading?

The klibc distribution contains some of the necessary software to make early userspace useful. The klibc distribution is currently maintained separately from the kernel.

You can obtain somewhat infrequent snapshots of klibc from https://www.kernel.org/pub/linux/libs/klibc/

For active users, you are better off using the klibc git repository, at http://git.kernel.org/?p=libs/klibc/klibc.git

The standalone klibc distribution currently provides three components, in addition to the klibc library:

- ipconfig, a program that configures network interfaces. It can configure them statically, or use DHCP to obtain information dynamically (aka "IP autoconfiguration").
- nfsmount, a program that can mount an NFS filesystem.
- kinit, the "glue" that uses ipconfig and nfsmount to replace the old support for IP autoconfig, mount a filesystem over NFS, and continue system boot using that filesystem as root.

kinit is built as a single statically linked binary to save space.

Eventually, several more chunks of kernel functionality will hopefully move to early userspace:

- Almost all of init/do_mounts* (the beginning of this is already in place)
- ACPI table parsing
- Insert unwieldy subsystem that doesn't really need to be in kernel space here

If kinit doesn't meet your current needs and you've got bytes to burn, the klibc distribution includes a small Bourne-compatible shell (ash) and a number of other utilities, so you can replace kinit and build custom initramfs images that meet your needs exactly.

For questions and help, you can sign up for the early userspace mailing list at http://www.zytor.com/mailman/listinfo/klibc

How does it work?

The kernel has currently 3 ways to mount the root filesystem:

a) all required device and filesystem drivers compiled into the kernel, no initrd. init/main.c:init() will call prepare_namespace() to mount the final root filesystem, based on the root= option and optional init= to run some other init binary than listed at the end of init/main.c:init().

b) some device and filesystem drivers built as modules and stored in an initrd. The initrd must contain a binary '/linuxrc' which is supposed to load these driver modules. It is also possible to mount the final root filesystem via linuxrc and use the pivot_root syscall. The initrd is mounted and executed via prepare_namespace().

c) using initramfs. The call to prepare_namespace() must be skipped. This means that a binary must do all the work. Said binary can be stored into initramfs either via modifying usr/gen_init_cpio.c or via the new initrd format, an cpio archive. It must be called "/init". This binary is responsible to do all the things prepare_namespace() would do.

   To maintain backwards compatibility, the /init binary will only run if it comes via an initramfs cpio archive. If this is not the case, init/main.c:init() will run prepare_namespace() to mount the final root and exec one of the predefined init binaries.

Bryan O'Sullivan <bos@serpentine.com>

5.2 initramfs buffer format

Al Viro, H. Peter Anvin

Last revision: 2002-01-13

Starting with kernel 2.5.x, the old "initial ramdisk" protocol is getting {replaced/complemented} with the new "initial ramfs" (initramfs) protocol. The initramfs contents is passed using the same memory buffer protocol used by the initrd protocol, but the contents is different. The initramfs buffer contains an archive which is expanded into a ramfs filesystem; this document details the format of the initramfs buffer format.

The initramfs buffer format is based around the "newc" or "crc" CPIO formats, and can be created with the cpio(1) utility. The cpio archive can be compressed using gzip(1). One valid version of an initramfs buffer is thus a single .cpio.gz file.

The full format of the initramfs buffer is defined by the following grammar, where:

```
*       is used to indicate "0 or more occurrences of"
(|)     indicates alternatives
+       indicates concatenation
GZIP()  indicates the gzip(1) of the operand
ALGN(n) means padding with null bytes to an n-byte boundary

initramfs  := ("\0" | cpio_archive | cpio_gzip_archive)*

cpio_gzip_archive := GZIP(cpio_archive)

cpio_archive := cpio_file* + (<nothing> | cpio_trailer)

cpio_file := ALGN(4) + cpio_header + filename + "\0" + ALGN(4) + data

cpio_trailer := ALGN(4) + cpio_header + "TRAILER!!!\0" + ALGN(4)
```

In human terms, the initramfs buffer contains a collection of compressed and/or uncompressed cpio archives (in the "newc" or "crc" formats); arbitrary amounts zero bytes (for padding) can be added between members.

The cpio "TRAILER!!!" entry (cpio end-of-archive) is optional, but is not ignored; see "handling of hard links" below.

The structure of the cpio_header is as follows (all fields contain hexadecimal ASCII numbers fully padded with '0' on the left to the full width of the field, for example, the integer 4780 is represented by the ASCII string "000012ac"):

Field name	Field size	Meaning
c_magic	6 bytes	The string “070701” or “070702”
c_ino	8 bytes	File inode number
c_mode	8 bytes	File mode and permissions
c_uid	8 bytes	File uid
c_gid	8 bytes	File gid
c_nlink	8 bytes	Number of links
c_mtime	8 bytes	Modification time
c_filesize	8 bytes	Size of data field
c_maj	8 bytes	Major part of file device number
c_min	8 bytes	Minor part of file device number
c_rmaj	8 bytes	Major part of device node reference
c_rmin	8 bytes	Minor part of device node reference
c_namesize	8 bytes	Length of filename, including final 0
c_chksum	8 bytes	Checksum of data field if c_magic is 070702; otherwise zero

The c_mode field matches the contents of st_mode returned by stat(2) on Linux, and encodes the file type and file permissions.

The c_filesize should be zero for any file which is not a regular file or symlink.

The c_chksum field contains a simple 32-bit unsigned sum of all the bytes in the data field. cpio(1) refers to this as “crc”, which is clearly incorrect (a cyclic redundancy check is a different and significantly stronger integrity check), however, this is the algorithm used.

If the filename is “TRAILER!!!” this is actually an end-of-archive marker; the c_filesize for an end-of-archive marker must be zero.

5.2.1 Handling of hard links

When a nondirectory with c_nlink > 1 is seen, the (c_maj,c_min,c_ino) tuple is looked up in a tuple buffer. If not found, it is entered in the tuple buffer and the entry is created as usual; if found, a hard link rather than a second copy of the file is created. It is not necessary (but permitted) to include a second copy of the file contents; if the file contents is not included, the c_filesize field should be set to zero to indicate no data section follows. If data is present, the previous instance of the file is overwritten; this allows the data-carrying instance of a file to occur anywhere in the sequence (GNU cpio is reported to attach the data to the last instance of a file only.)

c_filesize must not be zero for a symlink.

When a “TRAILER!!!” end-of-archive marker is seen, the tuple buffer is reset. This permits archives which are generated independently to be concatenated.

To combine file data from different sources (without having to regenerate the (c_maj,c_min,c_ino) fields), therefore, either one of the following techniques can be used:

a) Separate the different file data sources with a “TRAILER!!!” end-of-archive marker, or

b) Make sure c_nlink == 1 for all nondirectory entries.

CHAPTER

SIX

CPU AND DEVICE POWER MANAGEMENT

6.1 CPU Idle Time Management

Author Rafael J. Wysocki <rafael.j.wysocki@intel.com>

6.1.1 CPU Idle Time Management Subsystem

Every time one of the logical CPUs in the system (the entities that appear to fetch and execute instructions: hardware threads, if present, or processor cores) is idle after an interrupt or equivalent wakeup event, which means that there are no tasks to run on it except for the special "idle" task associated with it, there is an opportunity to save energy for the processor that it belongs to. That can be done by making the idle logical CPU stop fetching instructions from memory and putting some of the processor's functional units depended on by it into an idle state in which they will draw less power.

However, there may be multiple different idle states that can be used in such a situation in principle, so it may be necessary to find the most suitable one (from the kernel perspective) and ask the processor to use (or "enter") that particular idle state. That is the role of the CPU idle time management subsystem in the kernel, called `CPUIdle`.

The design of `CPUIdle` is modular and based on the code duplication avoidance principle, so the generic code that in principle need not depend on the hardware or platform design details in it is separate from the code that interacts with the hardware. It generally is divided into three categories of functional units: governors responsible for selecting idle states to ask the processor to enter, drivers that pass the governors' decisions on to the hardware and the core providing a common framework for them.

6.1.2 CPU Idle Time Governors

A CPU idle time (CPUIdle) governor is a bundle of policy code invoked when one of the logical CPUs in the system turns out to be idle. Its role is to select an idle state to ask the processor to enter in order to save some energy.

CPUIdle governors are generic and each of them can be used on any hardware platform that the Linux kernel can run on. For this reason, data structures operated on by them cannot depend on any hardware architecture or platform design details as well.

The governor itself is represented by a struct cpuidle_governor object containing four callback pointers, enable, disable, select, reflect, a rating field described below, and a name (string) used for identifying it.

For the governor to be available at all, that object needs to be registered with the CPUIdle core by calling cpuidle_register_governor() with a pointer to it passed as the argument. If successful, that causes the core to add the governor to the global list of available governors and, if it is the only one in the list (that is, the list was empty before) or the value of its rating field is greater than the value of that field for the governor currently in use, or the name of the new governor was passed to the kernel as the value of the cpuidle.governor= command line parameter, the new governor will be used from that point on (there can be only one CPUIdle governor in use at a time). Also, user space can choose the CPUIdle governor to use at run time via sysfs.

Once registered, CPUIdle governors cannot be unregistered, so it is not practical to put them into loadable kernel modules.

The interface between CPUIdle governors and the core consists of four callbacks:

enable

```
int (*enable) (struct cpuidle_driver *drv, struct cpuidle_device␣
↪*dev);
```

The role of this callback is to prepare the governor for handling the (logical) CPU represented by the struct cpuidle_device object pointed to by the dev argument. The struct cpuidle_driver object pointed to by the drv argument represents the CPUIdle driver to be used with that CPU (among other things, it should contain the list of struct cpuidle_state objects representing idle states that the processor holding the given CPU can be asked to enter).

It may fail, in which case it is expected to return a negative error code, and that causes the kernel to run the architecture-specific default code for idle CPUs on the CPU in question instead of CPUIdle until the ->enable() governor callback is invoked for that CPU again.

disable

```
void (*disable) (struct cpuidle_driver *drv, struct cpuidle_device␣
↪*dev);
```

Called to make the governor stop handling the (logical) CPU represented by the struct cpuidle_device object pointed to by the dev argument.

It is expected to reverse any changes made by the `->enable()` callback when it was last invoked for the target CPU, free all memory allocated by that callback and so on.

select

```
int (*select) (struct cpuidle_driver *drv, struct cpuidle_device *dev,
               bool *stop_tick);
```

Called to select an idle state for the processor holding the (logical) CPU represented by the `struct cpuidle_device` object pointed to by the `dev` argument.

The list of idle states to take into consideration is represented by the `states` array of `struct cpuidle_state` objects held by the `struct cpuidle_driver` object pointed to by the `drv` argument (which represents the `CPUIdle` driver to be used with the CPU at hand). The value returned by this callback is interpreted as an index into that array (unless it is a negative error code).

The `stop_tick` argument is used to indicate whether or not to stop the scheduler tick before asking the processor to enter the selected idle state. When the `bool` variable pointed to by it (which is set to `true` before invoking this callback) is cleared to `false`, the processor will be asked to enter the selected idle state without stopping the scheduler tick on the given CPU (if the tick has been stopped on that CPU already, however, it will not be restarted before asking the processor to enter the idle state).

This callback is mandatory (i.e. the `select` callback pointer in `struct cpuidle_governor` must not be `NULL` for the registration of the governor to succeed).

reflect

```
void (*reflect) (struct cpuidle_device *dev, int index);
```

Called to allow the governor to evaluate the accuracy of the idle state selection made by the `->select()` callback (when it was invoked last time) and possibly use the result of that to improve the accuracy of idle state selections in the future.

In addition, `CPUIdle` governors are required to take power management quality of service (PM QoS) constraints on the processor wakeup latency into account when selecting idle states. In order to obtain the current effective PM QoS wakeup latency constraint for a given CPU, a `CPUIdle` governor is expected to pass the number of the CPU to `cpuidle_governor_latency_req()`. Then, the governor's `->select()` callback must not return the index of an indle state whose `exit_latency` value is greater than the number returned by that function.

6.1.3 CPU Idle Time Management Drivers

CPU idle time management (CPUIdle) drivers provide an interface between the other parts of CPUIdle and the hardware.

First of all, a CPUIdle driver has to populate the states array of struct cpuidle_state objects included in the struct cpuidle_driver object representing it. Going forward this array will represent the list of available idle states that the processor hardware can be asked to enter shared by all of the logical CPUs handled by the given driver.

The entries in the states array are expected to be sorted by the value of the target_residency field in struct cpuidle_state in the ascending order (that is, index 0 should correspond to the idle state with the minimum value of target_residency). [Since the target_residency value is expected to reflect the "depth" of the idle state represented by the struct cpuidle_state object holding it, this sorting order should be the same as the ascending sorting order by the idle state "depth".]

Three fields in struct cpuidle_state are used by the existing CPUIdle governors for computations related to idle state selection:

target_residency Minimum time to spend in this idle state including the time needed to enter it (which may be substantial) to save more energy than could be saved by staying in a shallower idle state for the same amount of time, in microseconds.

exit_latency Maximum time it will take a CPU asking the processor to enter this idle state to start executing the first instruction after a wakeup from it, in microseconds.

flags Flags representing idle state properties. Currently, governors only use the CPUIDLE_FLAG_POLLING flag which is set if the given object does not represent a real idle state, but an interface to a software "loop" that can be used in order to avoid asking the processor to enter any idle state at all. [There are other flags used by the CPUIdle core in special situations.]

The enter callback pointer in struct cpuidle_state, which must not be NULL, points to the routine to execute in order to ask the processor to enter this particular idle state:

```
void (*enter) (struct cpuidle_device *dev, struct cpuidle_driver *drv,
               int index);
```

The first two arguments of it point to the struct cpuidle_device object representing the logical CPU running this callback and the struct cpuidle_driver object representing the driver itself, respectively, and the last one is an index of the struct cpuidle_state entry in the driver's states array representing the idle state to ask the processor to enter.

The analogous ->enter_s2idle() callback in struct cpuidle_state is used only for implementing the suspend-to-idle system-wide power management feature. The difference between in and ->enter() is that it must not re-enable interrupts at any point (even temporarily) or attempt to change the states of clock event devices, which the ->enter() callback may do sometimes.

Once the states array has been populated, the number of valid entries in it has to be stored in the state_count field of the struct cpuidle_driver object representing the driver. Moreover, if any entries in the states array represent "coupled"idle states (that is, idle states that can only be asked for if multiple related logical CPUs are idle), the safe_state_index field in struct cpuidle_driver needs to be the index of an idle state that is not "coupled" (that is, one that can be asked for if only one logical CPU is idle).

In addition to that, if the given CPUIdle driver is only going to handle a subset of logical CPUs in the system, the cpumask field in its struct cpuidle_driver object must point to the set (mask) of CPUs that will be handled by it.

A CPUIdle driver can only be used after it has been registered. If there are no "coupled" idle state entries in the driver's states array, that can be accomplished by passing the driver's struct cpuidle_driver object to cpuidle_register_driver(). Otherwise, cpuidle_register() should be used for this purpose.

However, it also is necessary to register struct cpuidle_device objects for all of the logical CPUs to be handled by the given CPUIdle driver with the help of cpuidle_register_device() after the driver has been registered and cpuidle_register_driver(), unlike cpuidle_register(), does not do that automatically. For this reason, the drivers that use cpuidle_register_driver() to register themselves must also take care of registering the struct cpuidle_device objects as needed, so it is generally recommended to use cpuidle_register() for CPUIdle driver registration in all cases.

The registration of a struct cpuidle_device object causes the CPUIdle sysfs interface to be created and the governor's ->enable() callback to be invoked for the logical CPU represented by it, so it must take place after registering the driver that will handle the CPU in question.

CPUIdle drivers and struct cpuidle_device objects can be unregistered when they are not necessary any more which allows some resources associated with them to be released. Due to dependencies between them, all of the struct cpuidle_device objects representing CPUs handled by the given CPUIdle driver must be unregistered, with the help of cpuidle_unregister_device(), before calling cpuidle_unregister_driver() to unregister the driver. Alternatively, cpuidle_unregister() can be called to unregister a CPUIdle driver along with all of the struct cpuidle_device objects representing CPUs handled by it.

CPUIdle drivers can respond to runtime system configuration changes that lead to modifications of the list of available processor idle states (which can happen, for example, when the system's power source is switched from AC to battery or the other way around). Upon a notification of such a change, a CPUIdle driver is expected to call cpuidle_pause_and_lock() to turn CPUIdle off temporarily and then cpuidle_disable_device() for all of the struct cpuidle_device objects representing CPUs affected by that change. Next, it can update its states array in accordance with the new configuration of the system, call cpuidle_enable_device() for all of the relevant struct cpuidle_device objects and invoke cpuidle_resume_and_unlock() to allow CPUIdle to be used again.

6.2 Device Power Management Basics

Author Rafael J. Wysocki <rafael.j.wysocki@intel.com>

Most of the code in Linux is device drivers, so most of the Linux power management (PM) code is also driver-specific. Most drivers will do very little; others, especially for platforms with small batteries (like cell phones), will do a lot.

This writeup gives an overview of how drivers interact with system-wide power management goals, emphasizing the models and interfaces that are shared by everything that hooks up to the driver model core. Read it as background for the domain-specific work you'd do with any specific driver.

6.2.1 Two Models for Device Power Management

Drivers will use one or both of these models to put devices into low-power states:

System Sleep model:

Drivers can enter low-power states as part of entering system-wide low-power states like "suspend" (also known as "suspend-to-RAM"), or (mostly for systems with disks) "hibernation" (also known as "suspend-to-disk").

This is something that device, bus, and class drivers collaborate on by implementing various role-specific suspend and resume methods to cleanly power down hardware and software subsystems, then reactivate them without loss of data.

Some drivers can manage hardware wakeup events, which make the system leave the low-power state. This feature may be enabled or disabled using the relevant `/sys/devices/.../power/wakeup` file (for Ethernet drivers the ioctl interface used by ethtool may also be used for this purpose); enabling it may cost some power usage, but let the whole system enter low-power states more often.

Runtime Power Management model:

Devices may also be put into low-power states while the system is running, independently of other power management activity in principle. However, devices are not generally independent of each other (for example, a parent device cannot be suspended unless all of its child devices have been suspended). Moreover, depending on the bus type the device is on, it may be necessary to carry out some bus-specific operations on the device for this purpose. Devices put into low power states at run time may require special handling during system-wide power transitions (suspend or hibernation).

> For these reasons not only the device driver itself, but also the appropriate subsystem (bus type, device type or device class) driver and the PM core are involved in runtime power management. As in the system sleep power management case, they need to collaborate by implementing various role-specific suspend and resume methods, so that the hardware is cleanly powered down and reactivated without data or service loss.

There's not a lot to be said about those low-power states except that they are very system-specific, and often device-specific. Also, that if enough devices have been put into low-power states (at runtime), the effect may be very similar to entering some system-wide low-power state (system sleep) ···and that synergies exist, so that several drivers using runtime PM might put the system into a state where even deeper power saving options are available.

Most suspended devices will have quiesced all I/O: no more DMA or IRQs (except for wakeup events), no more data read or written, and requests from upstream drivers are no longer accepted. A given bus or platform may have different requirements though.

Examples of hardware wakeup events include an alarm from a real time clock, network wake-on-LAN packets, keyboard or mouse activity, and media insertion or removal (for PCMCIA, MMC/SD, USB, and so on).

6.2.2 Interfaces for Entering System Sleep States

There are programming interfaces provided for subsystems (bus type, device type, device class) and device drivers to allow them to participate in the power management of devices they are concerned with. These interfaces cover both system sleep and runtime power management.

Device Power Management Operations

Device power management operations, at the subsystem level as well as at the device driver level, are implemented by defining and populating objects of type `struct dev_pm_ops` defined in `include/linux/pm.h`. The roles of the methods included in it will be explained in what follows. For now, it should be sufficient to remember that the last three methods are specific to runtime power management while the remaining ones are used during system-wide power transitions.

There also is a deprecated "old" or "legacy" interface for power management operations available at least for some subsystems. This approach does not use `struct dev_pm_ops` objects and it is suitable only for implementing system sleep power management methods in a limited way. Therefore it is not described in this document, so please refer directly to the source code for more information about it.

Subsystem-Level Methods

The core methods to suspend and resume devices reside in `struct dev_pm_ops` pointed to by the `ops` member of `struct dev_pm_domain`, or by the `pm` member of `struct bus_type`, `struct device_type` and `struct class`. They are mostly of interest to the people writing infrastructure for platforms and buses, like PCI or USB, or device type and device class drivers. They also are relevant to the writers of device drivers whose subsystems (PM domains, device types, device classes and bus types) don't provide all power management methods.

Bus drivers implement these methods as appropriate for the hardware and the drivers using it; PCI works differently from USB, and so on. Not many people write subsystem-level drivers; most driver code is a "device driver" that builds on top of bus-specific framework code.

For more information on these driver calls, see the description later; they are called in phases for every device, respecting the parent-child sequencing in the driver model tree.

`/sys/devices/.../power/wakeup` files

All device objects in the driver model contain fields that control the handling of system wakeup events (hardware signals that can force the system out of a sleep state). These fields are initialized by bus or device driver code using `device_set_wakeup_capable()` and `device_set_wakeup_enable()`, defined in `include/linux/pm_wakeup.h`.

The `power.can_wakeup` flag just records whether the device (and its driver) can physically support wakeup events. The `device_set_wakeup_capable()` routine affects this flag. The `power.wakeup` field is a pointer to an object of type `struct wakeup_source` used for controlling whether or not the device should use its system wakeup mechanism and for notifying the PM core of system wakeup events signaled by the device. This object is only present for wakeup-capable devices (i.e. devices whose `can_wakeup` flags are set) and is created (or removed) by `device_set_wakeup_capable()`.

Whether or not a device is capable of issuing wakeup events is a hardware matter, and the kernel is responsible for keeping track of it. By contrast, whether or not a wakeup-capable device should issue wakeup events is a policy decision, and it is managed by user space through a sysfs attribute: the `power/wakeup` file. User space can write the "enabled" or "disabled" strings to it to indicate whether or not, respectively, the device is supposed to signal system wakeup. This file is only present if the `power.wakeup` object exists for the given device and is created (or removed) along with that object, by `device_set_wakeup_capable()`. Reads from the file will return the corresponding string.

The initial value in the `power/wakeup` file is "disabled" for the majority of devices; the major exceptions are power buttons, keyboards, and Ethernet adapters whose WoL (wake-on-LAN) feature has been set up with ethtool. It should also default to "enabled" for devices that don't generate wakeup requests on their own but merely forward wakeup requests from one bus to another (like PCI Express ports).

The `device_may_wakeup()` routine returns true only if the `power.wakeup` object exists and the corresponding `power/wakeup` file contains the "enabled" string. This

information is used by subsystems, like the PCI bus type code, to see whether or not to enable the devices' wakeup mechanisms. If device wakeup mechanisms are enabled or disabled directly by drivers, they also should use `device_may_wakeup()` to decide what to do during a system sleep transition. Device drivers, however, are not expected to call `device_set_wakeup_enable()` directly in any case.

It ought to be noted that system wakeup is conceptually different from "remote wakeup" used by runtime power management, although it may be supported by the same physical mechanism. Remote wakeup is a feature allowing devices in low-power states to trigger specific interrupts to signal conditions in which they should be put into the full-power state. Those interrupts may or may not be used to signal system wakeup events, depending on the hardware design. On some systems it is impossible to trigger them from system sleep states. In any case, remote wakeup should always be enabled for runtime power management for all devices and drivers that support it.

`/sys/devices/.../power/control` files

Each device in the driver model has a flag to control whether it is subject to runtime power management. This flag, `runtime_auto`, is initialized by the bus type (or generally subsystem) code using `pm_runtime_allow()` or `pm_runtime_forbid()`; the default is to allow runtime power management.

The setting can be adjusted by user space by writing either "on" or "auto" to the device's `power/control` sysfs file. Writing "auto" calls `pm_runtime_allow()`, setting the flag and allowing the device to be runtime power-managed by its driver. Writing "on" calls `pm_runtime_forbid()`, clearing the flag, returning the device to full power if it was in a low-power state, and preventing the device from being runtime power-managed. User space can check the current value of the `runtime_auto` flag by reading that file.

The device's `runtime_auto` flag has no effect on the handling of system-wide power transitions. In particular, the device can (and in the majority of cases should and will) be put into a low-power state during a system-wide transition to a sleep state even though its `runtime_auto` flag is clear.

For more information about the runtime power management framework, refer to `Documentation/power/runtime_pm.rst`.

6.2.3 Calling Drivers to Enter and Leave System Sleep States

When the system goes into a sleep state, each device's driver is asked to suspend the device by putting it into a state compatible with the target system state. That's usually some version of "off", but the details are system-specific. Also, wakeup-enabled devices will usually stay partly functional in order to wake the system.

When the system leaves that low-power state, the device's driver is asked to resume it by returning it to full power. The suspend and resume operations always go together, and both are multi-phase operations.

For simple drivers, suspend might quiesce the device using class code and then turn its hardware as "off" as possible during suspend_noirq. The matching resume

calls would then completely reinitialize the hardware before reactivating its class I/O queues.

More power-aware drivers might prepare the devices for triggering system wakeup events.

Call Sequence Guarantees

To ensure that bridges and similar links needing to talk to a device are available when the device is suspended or resumed, the device hierarchy is walked in a bottom-up order to suspend devices. A top-down order is used to resume those devices.

The ordering of the device hierarchy is defined by the order in which devices get registered: a child can never be registered, probed or resumed before its parent; and can't be removed or suspended after that parent.

The policy is that the device hierarchy should match hardware bus topology. [Or at least the control bus, for devices which use multiple busses.] In particular, this means that a device registration may fail if the parent of the device is suspending (i.e. has been chosen by the PM core as the next device to suspend) or has already suspended, as well as after all of the other devices have been suspended. Device drivers must be prepared to cope with such situations.

System Power Management Phases

Suspending or resuming the system is done in several phases. Different phases are used for suspend-to-idle, shallow (standby), and deep ("suspend-to-RAM") sleep states and the hibernation state ("suspend-to-disk"). Each phase involves executing callbacks for every device before the next phase begins. Not all buses or classes support all these callbacks and not all drivers use all the callbacks. The various phases always run after tasks have been frozen and before they are unfrozen. Furthermore, the `*_noirq` phases run at a time when IRQ handlers have been disabled (except for those marked with the IRQF_NO_SUSPEND flag).

All phases use PM domain, bus, type, class or driver callbacks (that is, methods defined in `dev->pm_domain->ops`, `dev->bus->pm`, `dev->type->pm`, `dev->class->pm` or `dev->driver->pm`). These callbacks are regarded by the PM core as mutually exclusive. Moreover, PM domain callbacks always take precedence over all of the other callbacks and, for example, type callbacks take precedence over bus, class and driver callbacks. To be precise, the following rules are used to determine which callback to execute in the given phase:

1. If `dev->pm_domain` is present, the PM core will choose the callback provided by `dev->pm_domain->ops` for execution.
2. Otherwise, if both `dev->type` and `dev->type->pm` are present, the callback provided by `dev->type->pm` will be chosen for execution.
3. Otherwise, if both `dev->class` and `dev->class->pm` are present, the callback provided by `dev->class->pm` will be chosen for execution.
4. Otherwise, if both `dev->bus` and `dev->bus->pm` are present, the callback provided by `dev->bus->pm` will be chosen for execution.

This allows PM domains and device types to override callbacks provided by bus types or device classes if necessary.

The PM domain, type, class and bus callbacks may in turn invoke device- or driver-specific methods stored in `dev->driver->pm`, but they don't have to do that.

If the subsystem callback chosen for execution is not present, the PM core will execute the corresponding method from the `dev->driver->pm` set instead if there is one.

Entering System Suspend

When the system goes into the freeze, standby or memory sleep state, the phases are: `prepare`, `suspend`, `suspend_late`, `suspend_noirq`.

1. The `prepare` phase is meant to prevent races by preventing new devices from being registered; the PM core would never know that all the children of a device had been suspended if new children could be registered at will. [By contrast, from the PM core's perspective, devices may be unregistered at any time.] Unlike the other suspend-related phases, during the `prepare` phase the device hierarchy is traversed top-down.

   After the `->prepare` callback method returns, no new children may be registered below the device. The method may also prepare the device or driver in some way for the upcoming system power transition, but it should not put the device into a low-power state. Moreover, if the device supports runtime power management, the `->prepare` callback method must not update its state in case it is necessary to resume it from runtime suspend later on.

   For devices supporting runtime power management, the return value of the prepare callback can be used to indicate to the PM core that it may safely leave the device in runtime suspend (if runtime-suspended already), provided that all of the device's descendants are also left in runtime suspend. Namely, if the prepare callback returns a positive number and that happens for all of the descendants of the device too, and all of them (including the device itself) are runtime-suspended, the PM core will skip the `suspend`, `suspend_late` and `suspend_noirq` phases as well as all of the corresponding phases of the subsequent device resume for all of these devices. In that case, the `->complete` callback will be the next one invoked after the `->prepare` callback and is entirely responsible for putting the device into a consistent state as appropriate.

   Note that this direct-complete procedure applies even if the device is disabled for runtime PM; only the runtime-PM status matters. It follows that if a device has system-sleep callbacks but does not support runtime PM, then its prepare callback must never return a positive value. This is because all such devices are initially set to runtime-suspended with runtime PM disabled.

   This feature also can be controlled by device drivers by using the `DPM_FLAG_NO_DIRECT_COMPLETE` and `DPM_FLAG_SMART_PREPARE` driver power management flags. [Typically, they are set at the time the driver is probed against the device in question by passing them to the `dev_pm_set_driver_flags()` helper function.] If the first of these flags is set, the PM core will not apply the direct-complete procedure described above to the given device and, consequenty, to any of its ancestors. The second flag,

when set, informs the middle layer code (bus types, device types, PM domains, classes) that it should take the return value of the `->prepare` callback provided by the driver into account and it may only return a positive value from its own `->prepare` callback if the driver's one also has returned a positive value.

2. The `->suspend` methods should quiesce the device to stop it from performing I/O. They also may save the device registers and put it into the appropriate low-power state, depending on the bus type the device is on, and they may enable wakeup events.

   However, for devices supporting runtime power management, the `->suspend` methods provided by subsystems (bus types and PM domains in particular) must follow an additional rule regarding what can be done to the devices before their drivers' `->suspend` methods are called. Namely, they may resume the devices from runtime suspend by calling `pm_runtime_resume()` for them, if that is necessary, but they must not update the state of the devices in any other way at that time (in case the drivers need to resume the devices from runtime suspend in their `->suspend` methods). In fact, the PM core prevents subsystems or drivers from putting devices into runtime suspend at these times by calling `pm_runtime_get_noresume()` before issuing the `->prepare` callback (and calling `pm_runtime_put()` after issuing the `->complete` callback).

3. For a number of devices it is convenient to split suspend into the "quiesce device" and "save device state" phases, in which cases `suspend_late` is meant to do the latter. It is always executed after runtime power management has been disabled for the device in question.

4. The `suspend_noirq` phase occurs after IRQ handlers have been disabled, which means that the driver's interrupt handler will not be called while the callback method is running. The `->suspend_noirq` methods should save the values of the device's registers that weren't saved previously and finally put the device into the appropriate low-power state.

   The majority of subsystems and device drivers need not implement this callback. However, bus types allowing devices to share interrupt vectors, like PCI, generally need it; otherwise a driver might encounter an error during the suspend phase by fielding a shared interrupt generated by some other device after its own device had been set to low power.

At the end of these phases, drivers should have stopped all I/O transactions (DMA, IRQs), saved enough state that they can re-initialize or restore previous state (as needed by the hardware), and placed the device into a low-power state. On many platforms they will gate off one or more clock sources; sometimes they will also switch off power supplies or reduce voltages. [Drivers supporting runtime PM may already have performed some or all of these steps.]

If `device_may_wakeup(dev)()` returns `true`, the device should be prepared for generating hardware wakeup signals to trigger a system wakeup event when the system is in the sleep state. For example, `enable_irq_wake()` might identify GPIO signals hooked up to a switch or other external hardware, and `pci_enable_wake()` does something similar for the PCI PME signal.

If any of these callbacks returns an error, the system won't enter the desired

low-power state. Instead, the PM core will unwind its actions by resuming all the devices that were suspended.

Leaving System Suspend

When resuming from freeze, standby or memory sleep, the phases are: `resume_noirq`, `resume_early`, `resume`, `complete`.

1. The `->resume_noirq` callback methods should perform any actions needed before the driver's interrupt handlers are invoked. This generally means undoing the actions of the `suspend_noirq` phase. If the bus type permits devices to share interrupt vectors, like PCI, the method should bring the device and its driver into a state in which the driver can recognize if the device is the source of incoming interrupts, if any, and handle them correctly.

   For example, the PCI bus type's `->pm.resume_noirq()` puts the device into the full-power state (D0 in the PCI terminology) and restores the standard configuration registers of the device. Then it calls the device driver's `->pm.resume_noirq()` method to perform device-specific actions.

2. The `->resume_early` methods should prepare devices for the execution of the resume methods. This generally involves undoing the actions of the preceding `suspend_late` phase.

3. The `->resume` methods should bring the device back to its operating state, so that it can perform normal I/O. This generally involves undoing the actions of the `suspend` phase.

4. The `complete` phase should undo the actions of the `prepare` phase. For this reason, unlike the other resume-related phases, during the `complete` phase the device hierarchy is traversed bottom-up.

   Note, however, that new children may be registered below the device as soon as the `->resume` callbacks occur; it's not necessary to wait until the `complete` phase runs.

   Moreover, if the preceding `->prepare` callback returned a positive number, the device may have been left in runtime suspend throughout the whole system suspend and resume (its `->suspend`, `->suspend_late`, `->suspend_noirq`, `->resume_noirq`, `->resume_early`, and `->resume` callbacks may have been skipped). In that case, the `->complete` callback is entirely responsible for putting the device into a consistent state after system suspend if necessary. [For example, it may need to queue up a runtime resume request for the device for this purpose.] To check if that is the case, the `->complete` callback can consult the device's `power.direct_complete` flag. If that flag is set when the `->complete` callback is being run then the direct-complete mechanism was used, and special actions may be required to make the device work correctly afterward.

At the end of these phases, drivers should be as functional as they were before suspending: I/O can be performed using DMA and IRQs, and the relevant clocks are gated on.

However, the details here may again be platform-specific. For example, some systems support multiple "run" states, and the mode in effect at the end of re-

sume might not be the one which preceded suspension. That means availability of certain clocks or power supplies changed, which could easily affect how a driver works.

Drivers need to be able to handle hardware which has been reset since all of the suspend methods were called, for example by complete reinitialization. This may be the hardest part, and the one most protected by NDA'd documents and chip errata. It's simplest if the hardware state hasn't changed since the suspend was carried out, but that can only be guaranteed if the target system sleep entered was suspend-to-idle. For the other system sleep states that may not be the case (and usually isn't for ACPI-defined system sleep states, like S3).

Drivers must also be prepared to notice that the device has been removed while the system was powered down, whenever that's physically possible. PCMCIA, MMC, USB, Firewire, SCSI, and even IDE are common examples of busses where common Linux platforms will see such removal. Details of how drivers will notice and handle such removals are currently bus-specific, and often involve a separate thread.

These callbacks may return an error value, but the PM core will ignore such errors since there's nothing it can do about them other than printing them in the system log.

Entering Hibernation

Hibernating the system is more complicated than putting it into sleep states, because it involves creating and saving a system image. Therefore there are more phases for hibernation, with a different set of callbacks. These phases always run after tasks have been frozen and enough memory has been freed.

The general procedure for hibernation is to quiesce all devices ("freeze"), create an image of the system memory while everything is stable, reactivate all devices ("thaw"), write the image to permanent storage, and finally shut down the system ("power off"). The phases used to accomplish this are: `prepare`, `freeze`, `freeze_late`, `freeze_noirq`, `thaw_noirq`, `thaw_early`, `thaw`, `complete`, `prepare`, `poweroff`, `poweroff_late`, `poweroff_noirq`.

1. The `prepare` phase is discussed in the "Entering System Suspend" section above.
2. The `->freeze` methods should quiesce the device so that it doesn't generate IRQs or DMA, and they may need to save the values of device registers. However the device does not have to be put in a low-power state, and to save time it's best not to do so. Also, the device should not be prepared to generate wakeup events.
3. The `freeze_late` phase is analogous to the `suspend_late` phase described earlier, except that the device should not be put into a low-power state and should not be allowed to generate wakeup events.
4. The `freeze_noirq` phase is analogous to the `suspend_noirq` phase discussed earlier, except again that the device should not be put into a low-power state and should not be allowed to generate wakeup events.

At this point the system image is created. All devices should be inactive and the contents of memory should remain undisturbed while this happens, so that the image forms an atomic snapshot of the system state.

5. The `thaw_noirq` phase is analogous to the `resume_noirq` phase discussed earlier. The main difference is that its methods can assume the device is in the same state as at the end of the `freeze_noirq` phase.

6. The `thaw_early` phase is analogous to the `resume_early` phase described above. Its methods should undo the actions of the preceding `freeze_late`, if necessary.

7. The `thaw` phase is analogous to the `resume` phase discussed earlier. Its methods should bring the device back to an operating state, so that it can be used for saving the image if necessary.

8. The `complete` phase is discussed in the "Leaving System Suspend" section above.

At this point the system image is saved, and the devices then need to be prepared for the upcoming system shutdown. This is much like suspending them before putting the system into the suspend-to-idle, shallow or deep sleep state, and the phases are similar.

9. The `prepare` phase is discussed above.

10. The `poweroff` phase is analogous to the `suspend` phase.

11. The `poweroff_late` phase is analogous to the `suspend_late` phase.

12. The `poweroff_noirq` phase is analogous to the `suspend_noirq` phase.

The `->poweroff`, `->poweroff_late` and `->poweroff_noirq` callbacks should do essentially the same things as the `->suspend`, `->suspend_late` and `->suspend_noirq` callbacks, respectively. A notable difference is that they need not store the device register values, because the registers should already have been stored during the `freeze`, `freeze_late` or `freeze_noirq` phases. Also, on many machines the firmware will power-down the entire system, so it is not necessary for the callback to put the device in a low-power state.

Leaving Hibernation

Resuming from hibernation is, again, more complicated than resuming from a sleep state in which the contents of main memory are preserved, because it requires a system image to be loaded into memory and the pre-hibernation memory contents to be restored before control can be passed back to the image kernel.

Although in principle the image might be loaded into memory and the pre-hibernation memory contents restored by the boot loader, in practice this can't be done because boot loaders aren't smart enough and there is no established protocol for passing the necessary information. So instead, the boot loader loads a fresh instance of the kernel, called "the restore kernel", into memory and passes control to it in the usual way. Then the restore kernel reads the system image, restores the pre-hibernation memory contents, and passes control to the image kernel. Thus two different kernel instances are involved in resuming from hibernation. In fact, the restore kernel may be completely different from the image

kernel: a different configuration and even a different version. This has important consequences for device drivers and their subsystems.

To be able to load the system image into memory, the restore kernel needs to include at least a subset of device drivers allowing it to access the storage medium containing the image, although it doesn't need to include all of the drivers present in the image kernel. After the image has been loaded, the devices managed by the boot kernel need to be prepared for passing control back to the image kernel. This is very similar to the initial steps involved in creating a system image, and it is accomplished in the same way, using `prepare`, `freeze`, and `freeze_noirq` phases. However, the devices affected by these phases are only those having drivers in the restore kernel; other devices will still be in whatever state the boot loader left them.

Should the restoration of the pre-hibernation memory contents fail, the restore kernel would go through the "thawing" procedure described above, using the `thaw_noirq`, `thaw_early`, `thaw`, and `complete` phases, and then continue running normally. This happens only rarely. Most often the pre-hibernation memory contents are restored successfully and control is passed to the image kernel, which then becomes responsible for bringing the system back to the working state.

To achieve this, the image kernel must restore the devices' pre-hibernation functionality. The operation is much like waking up from a sleep state (with the memory contents preserved), although it involves different phases: `restore_noirq`, `restore_early`, `restore`, `complete`.

1. The `restore_noirq` phase is analogous to the `resume_noirq` phase.
2. The `restore_early` phase is analogous to the `resume_early` phase.
3. The `restore` phase is analogous to the `resume` phase.
4. The `complete` phase is discussed above.

The main difference from `resume[_early|_noirq]` is that `restore[_early|_noirq]` must assume the device has been accessed and reconfigured by the boot loader or the restore kernel. Consequently, the state of the device may be different from the state remembered from the `freeze`, `freeze_late` and `freeze_noirq` phases. The device may even need to be reset and completely re-initialized. In many cases this difference doesn't matter, so the `->resume[_early|_noirq]` and `->restore[_early|_norq]` method pointers can be set to the same routines. Nevertheless, different callback pointers are used in case there is a situation where it actually does matter.

6.2.4 Power Management Notifiers

There are some operations that cannot be carried out by the power management callbacks discussed above, because the callbacks occur too late or too early. To handle these cases, subsystems and device drivers may register power management notifiers that are called before tasks are frozen and after they have been thawed. Generally speaking, the PM notifiers are suitable for performing actions that either require user space to be available, or at least won't interfere with user space.

For details refer to Suspend/Hibernation Notifiers.

6.2.5 Device Low-Power (suspend) States

Device low-power states aren't standard. One device might only handle "on" and "off", while another might support a dozen different versions of "on" (how many engines are active?), plus a state that gets back to "on" faster than from a full "off".

Some buses define rules about what different suspend states mean. PCI gives one example: after the suspend sequence completes, a non-legacy PCI device may not perform DMA or issue IRQs, and any wakeup events it issues would be issued through the PME# bus signal. Plus, there are several PCI-standard device states, some of which are optional.

In contrast, integrated system-on-chip processors often use IRQs as the wakeup event sources (so drivers would call `enable_irq_wake()`) and might be able to treat DMA completion as a wakeup event (sometimes DMA can stay active too, it'd only be the CPU and some peripherals that sleep).

Some details here may be platform-specific. Systems may have devices that can be fully active in certain sleep states, such as an LCD display that's refreshed using DMA while most of the system is sleeping lightly ⋯and its frame buffer might even be updated by a DSP or other non-Linux CPU while the Linux control processor stays idle.

Moreover, the specific actions taken may depend on the target system state. One target system state might allow a given device to be very operational; another might require a hard shut down with re-initialization on resume. And two different target systems might use the same device in different ways; the aforementioned LCD might be active in one product's "standby", but a different product using the same SOC might work differently.

6.2.6 Device Power Management Domains

Sometimes devices share reference clocks or other power resources. In those cases it generally is not possible to put devices into low-power states individually. Instead, a set of devices sharing a power resource can be put into a low-power state together at the same time by turning off the shared power resource. Of course, they also need to be put into the full-power state together, by turning the shared power resource on. A set of devices with this property is often referred to as a power domain. A power domain may also be nested inside another power domain. The nested domain is referred to as the sub-domain of the parent domain.

Support for power domains is provided through the `pm_domain` field of `struct device`. This field is a pointer to an object of type `struct dev_pm_domain`, defined in `include/linux/pm.h`, providing a set of power management callbacks analogous to the subsystem-level and device driver callbacks that are executed for the given device during all power transitions, instead of the respective subsystem-level callbacks. Specifically, if a device's `pm_domain` pointer is not NULL, the `->suspend()` callback from the object pointed to by it will be executed instead of its subsystem's (e.g. bus type's) `->suspend()` callback and analogously for all of the remaining callbacks. In other words, power management domain callbacks, if defined for the given device, always take precedence over the callbacks provided by the device's subsystem (e.g. bus type).

The support for device power management domains is only relevant to platforms needing to use the same device driver power management callbacks in many different power domain configurations and wanting to avoid incorporating the support for power domains into subsystem-level callbacks, for example by modifying the platform bus type. Other platforms need not implement it or take it into account in any way.

Devices may be defined as IRQ-safe which indicates to the PM core that their runtime PM callbacks may be invoked with disabled interrupts (see `Documentation/power/runtime_pm.rst` for more information). If an IRQ-safe device belongs to a PM domain, the runtime PM of the domain will be disallowed, unless the domain itself is defined as IRQ-safe. However, it makes sense to define a PM domain as IRQ-safe only if all the devices in it are IRQ-safe. Moreover, if an IRQ-safe domain has a parent domain, the runtime PM of the parent is only allowed if the parent itself is IRQ-safe too with the additional restriction that all child domains of an IRQ-safe parent must also be IRQ-safe.

6.2.7 Runtime Power Management

Many devices are able to dynamically power down while the system is still running. This feature is useful for devices that are not being used, and can offer significant power savings on a running system. These devices often support a range of runtime power states, which might use names such as “off”, “sleep”, “idle”, “active”, and so on. Those states will in some cases (like PCI) be partially constrained by the bus the device uses, and will usually include hardware states that are also used in system sleep states.

A system-wide power transition can be started while some devices are in low power states due to runtime power management. The system sleep PM callbacks should recognize such situations and react to them appropriately, but the necessary actions are subsystem-specific.

In some cases the decision may be made at the subsystem level while in other cases the device driver may be left to decide. In some cases it may be desirable to leave a suspended device in that state during a system-wide power transition, but in other cases the device must be put back into the full-power state temporarily, for example so that its system wakeup capability can be disabled. This all depends on the hardware and the design of the subsystem and device driver in question.

If it is necessary to resume a device from runtime suspend during a system-wide transition into a sleep state, that can be done by calling `pm_runtime_resume()` from the `->suspend` callback (or the `->freeze` or `->poweroff` callback for transitions related to hibernation) of either the device’s driver or its subsystem (for example, a bus type or a PM domain). However, subsystems must not otherwise change the runtime status of devices from their `->prepare` and `->suspend` callbacks (or equivalent) before invoking device drivers’ `->suspend` callbacks (or equivalent).

The DPM_FLAG_SMART_SUSPEND Driver Flag

Some bus types and PM domains have a policy to resume all devices from runtime suspend upfront in their ->suspend callbacks, but that may not be really necessary if the device's driver can cope with runtime-suspended devices. The driver can indicate this by setting DPM_FLAG_SMART_SUSPEND in power.driver_flags at probe time, with the assistance of the dev_pm_set_driver_flags() helper routine.

Setting that flag causes the PM core and middle-layer code (bus types, PM domains etc.) to skip the ->suspend_late and ->suspend_noirq callbacks provided by the driver if the device remains in runtime suspend throughout those phases of the system-wide suspend (and similarly for the "freeze" and "poweroff" parts of system hibernation). [Otherwise the same driver callback might be executed twice in a row for the same device, which would not be valid in general.] If the middle-layer system-wide PM callbacks are present for the device then they are responsible for skipping these driver callbacks; if not then the PM core skips them. The subsystem callback routines can determine whether they need to skip the driver callbacks by testing the return value from the dev_pm_skip_suspend() helper function.

In addition, with DPM_FLAG_SMART_SUSPEND set, the driver's ->thaw_noirq and ->thaw_early callbacks are skipped in hibernation if the device remained in runtime suspend throughout the preceding "freeze" transition. Again, if the middle-layer callbacks are present for the device, they are responsible for doing this, otherwise the PM core takes care of it.

The DPM_FLAG_MAY_SKIP_RESUME Driver Flag

During system-wide resume from a sleep state it's easiest to put devices into the full-power state, as explained in Documentation/power/runtime_pm.rst. [Refer to that document for more information regarding this particular issue as well as for information on the device runtime power management framework in general.] However, it often is desirable to leave devices in suspend after system transitions to the working state, especially if those devices had been in runtime suspend before the preceding system-wide suspend (or analogous) transition.

To that end, device drivers can use the DPM_FLAG_MAY_SKIP_RESUME flag to indicate to the PM core and middle-layer code that they allow their "noirq" and "early" resume callbacks to be skipped if the device can be left in suspend after system-wide PM transitions to the working state. Whether or not that is the case generally depends on the state of the device before the given system suspend-resume cycle and on the type of the system transition under way. In particular, the "thaw" and "restore" transitions related to hibernation are not affected by DPM_FLAG_MAY_SKIP_RESUME at all. [All callbacks are issued during the "restore" transition regardless of the flag settings, and whether or not any driver callbacks are skipped during the "thaw" transition depends whether or not the DPM_FLAG_SMART_SUSPEND flag is set (see above). In addition, a device is not allowed to remain in runtime suspend if any of its children will be returned to full power.]

The DPM_FLAG_MAY_SKIP_RESUME flag is taken into account in combination with the power.may_skip_resume status bit set by the PM core during the "suspend" phase of suspend-type transitions. If the driver or the middle layer has a reason to prevent the driver's "noirq"and "early"resume callbacks from being skipped during

the subsequent system resume transition, it should clear power.may_skip_resume in its ->suspend, ->suspend_late or ->suspend_noirq callback. [Note that the drivers setting DPM_FLAG_SMART_SUSPEND need to clear power.may_skip_resume in their ->suspend callback in case the other two are skipped.]

Setting the power.may_skip_resume status bit along with the DPM_FLAG_MAY_SKIP_RESUME flag is necessary, but generally not sufficient, for the driver's "noirq" and "early" resume callbacks to be skipped. Whether or not they should be skipped can be determined by evaluating the dev_pm_skip_resume() helper function.

If that function returns true, the driver's "noirq" and "early" resume callbacks should be skipped and the device's runtime PM status will be set to "suspended" by the PM core. Otherwise, if the device was runtime-suspended during the preceding system-wide suspend transition and its DPM_FLAG_SMART_SUSPEND is set, its runtime PM status will be set to "active" by the PM core. [Hence, the drivers that do not set DPM_FLAG_SMART_SUSPEND should not expect the runtime PM status of their devices to be changed from "suspended" to "active" by the PM core during system-wide resume-type transitions.]

If the DPM_FLAG_MAY_SKIP_RESUME flag is not set for a device, but DPM_FLAG_SMART_SUSPEND is set and the driver's "late" and "noirq" suspend callbacks are skipped, its system-wide "noirq" and "early" resume callbacks, if present, are invoked as usual and the device's runtime PM status is set to "active" by the PM core before enabling runtime PM for it. In that case, the driver must be prepared to cope with the invocation of its system-wide resume callbacks back-to-back with its ->runtime_suspend one (without the intervening ->runtime_resume and system-wide suspend callbacks) and the final state of the device must reflect the "active" runtime PM status in that case. [Note that this is not a problem at all if the driver's ->suspend_late callback pointer points to the same function as its ->runtime_suspend one and its ->resume_early callback pointer points to the same function as the ->runtime_resume one, while none of the other system-wide suspend-resume callbacks of the driver are present, for example.]

Likewise, if DPM_FLAG_MAY_SKIP_RESUME is set for a device, its driver's system-wide "noirq"and "early"resume callbacks may be skipped while its "late"and "noirq" suspend callbacks may have been executed (in principle, regardless of whether or not DPM_FLAG_SMART_SUSPEND is set). In that case, the driver needs to be able to cope with the invocation of its ->runtime_resume callback back-to-back with its "late" and "noirq" suspend ones. [For instance, that is not a concern if the driver sets both DPM_FLAG_SMART_SUSPEND and DPM_FLAG_MAY_SKIP_RESUME and uses the same pair of suspend/resume callback functions for runtime PM and system-wide suspend/resume.]

6.3 Suspend/Hibernation Notifiers

Author Rafael J. Wysocki <rafael.j.wysocki@intel.com>

There are some operations that subsystems or drivers may want to carry out before hibernation/suspend or after restore/resume, but they require the system to be fully functional, so the drivers' and subsystems' `->suspend()` and `->resume()` or even `->prepare()` and `->complete()` callbacks are not suitable for this purpose.

For example, device drivers may want to upload firmware to their devices after resume/restore, but they cannot do it by calling `request_firmware()` from their `->resume()` or `->complete()` callback routines (user land processes are frozen at these points). The solution may be to load the firmware into memory before processes are frozen and upload it from there in the `->resume()` routine. A suspend/hibernation notifier may be used for that.

Subsystems or drivers having such needs can register suspend notifiers that will be called upon the following events by the PM core:

PM_HIBERNATION_PREPARE The system is going to hibernate, tasks will be frozen immediately. This is different from PM_SUSPEND_PREPARE below, because in this case additional work is done between the notifiers and the invocation of PM callbacks for the "freeze" transition.

PM_POST_HIBERNATION The system memory state has been restored from a hibernation image or an error occurred during hibernation. Device restore callbacks have been executed and tasks have been thawed.

PM_RESTORE_PREPARE The system is going to restore a hibernation image. If all goes well, the restored image kernel will issue a PM_POST_HIBERNATION notification.

PM_POST_RESTORE An error occurred during restore from hibernation. Device restore callbacks have been executed and tasks have been thawed.

PM_SUSPEND_PREPARE The system is preparing for suspend.

PM_POST_SUSPEND The system has just resumed or an error occurred during suspend. Device resume callbacks have been executed and tasks have been thawed.

It is generally assumed that whatever the notifiers do for PM_HIBERNATION_PREPARE, should be undone for PM_POST_HIBERNATION. Analogously, operations carried out for PM_SUSPEND_PREPARE should be reversed for PM_POST_SUSPEND.

Moreover, if one of the notifiers fails for the PM_HIBERNATION_PREPARE or PM_SUSPEND_PREPARE event, the notifiers that have already succeeded for that event will be called for PM_POST_HIBERNATION or PM_POST_SUSPEND, respectively.

The hibernation and suspend notifiers are called with pm_mutex held. They are defined in the usual way, but their last argument is meaningless (it is always NULL).

To register and/or unregister a suspend notifier use register_pm_notifier() and unregister_pm_notifier(), respectively (both defined in include/linux/

suspend.h). If you don't need to unregister the notifier, you can also use the pm_notifier() macro defined in include/linux/suspend.h.

6.4 Device Power Management Data Types

struct **dev_pm_ops**
device PM callbacks.

Definition

```
struct dev_pm_ops {
  int (*prepare)(struct device *dev);
  void (*complete)(struct device *dev);
  int (*suspend)(struct device *dev);
  int (*resume)(struct device *dev);
  int (*freeze)(struct device *dev);
  int (*thaw)(struct device *dev);
  int (*poweroff)(struct device *dev);
  int (*restore)(struct device *dev);
  int (*suspend_late)(struct device *dev);
  int (*resume_early)(struct device *dev);
  int (*freeze_late)(struct device *dev);
  int (*thaw_early)(struct device *dev);
  int (*poweroff_late)(struct device *dev);
  int (*restore_early)(struct device *dev);
  int (*suspend_noirq)(struct device *dev);
  int (*resume_noirq)(struct device *dev);
  int (*freeze_noirq)(struct device *dev);
  int (*thaw_noirq)(struct device *dev);
  int (*poweroff_noirq)(struct device *dev);
  int (*restore_noirq)(struct device *dev);
  int (*runtime_suspend)(struct device *dev);
  int (*runtime_resume)(struct device *dev);
  int (*runtime_idle)(struct device *dev);
};
```

Members

prepare The principal role of this callback is to prevent new children of the device from being registered after it has returned (the driver's subsystem and generally the rest of the kernel is supposed to prevent new calls to the probe method from being made too once **prepare()** has succeeded). If **prepare()** detects a situation it cannot handle (e.g. registration of a child already in progress), it may return -EAGAIN, so that the PM core can execute it once again (e.g. after a new child has been registered) to recover from the race condition. This method is executed for all kinds of suspend transitions and is followed by one of the suspend callbacks: **suspend()**, **freeze()**, or **poweroff()**. If the transition is a suspend to memory or standby (that is, not related to hibernation), the return value of **prepare()** may be used to indicate to the PM core to leave the device in runtime suspend if applicable. Namely, if **prepare()** returns a positive number, the PM core will understand that as a declaration that the device appears to be runtime-suspended and it may be left in that state during the entire transition and during the subsequent resume if all of its descendants are left in runtime suspend too. If that happens, **complete()** will be

executed directly after **prepare()** and it must ensure the proper functioning of the device after the system resume. The PM core executes subsystem-level **prepare()** for all devices before starting to invoke suspend callbacks for any of them, so generally devices may be assumed to be functional or to respond to runtime resume requests while **prepare()** is being executed. However, device drivers may NOT assume anything about the availability of user space at that time and it is NOT valid to request firmware from within **prepare()** (it' s too late to do that). It also is NOT valid to allocate substantial amounts of memory from **prepare()** in the GFP_KERNEL mode. [To work around these limitations, drivers may register suspend and hibernation notifiers to be executed before the freezing of tasks.]

complete Undo the changes made by **prepare()**. This method is executed for all kinds of resume transitions, following one of the resume callbacks: **resume()**, **thaw()**, **restore()**. Also called if the state transition fails before the driver' s suspend callback: **suspend()**, **freeze()** or **poweroff()**, can be executed (e.g. if the suspend callback fails for one of the other devices that the PM core has unsuccessfully attempted to suspend earlier). The PM core executes subsystem-level **complete()** after it has executed the appropriate resume callbacks for all devices. If the corresponding **prepare()** at the beginning of the suspend transition returned a positive number and the device was left in runtime suspend (without executing any suspend and resume callbacks for it), **complete()** will be the only callback executed for the device during resume. In that case, **complete()** must be prepared to do whatever is necessary to ensure the proper functioning of the device after the system resume. To this end, **complete()** can check the power.direct_complete flag of the device to learn whether (unset) or not (set) the previous suspend and resume callbacks have been executed for it.

suspend Executed before putting the system into a sleep state in which the contents of main memory are preserved. The exact action to perform depends on the device' s subsystem (PM domain, device type, class or bus type), but generally the device must be quiescent after subsystem-level **suspend()** has returned, so that it doesn' t do any I/O or DMA. Subsystem-level **suspend()** is executed for all devices after invoking subsystem-level **prepare()** for all of them.

resume Executed after waking the system up from a sleep state in which the contents of main memory were preserved. The exact action to perform depends on the device' s subsystem, but generally the driver is expected to start working again, responding to hardware events and software requests (the device itself may be left in a low-power state, waiting for a runtime resume to occur). The state of the device at the time its driver' s **resume()** callback is run depends on the platform and subsystem the device belongs to. On most platforms, there are no restrictions on availability of resources like clocks during **resume()**. Subsystem-level **resume()** is executed for all devices after invoking subsystem-level **resume_noirq()** for all of them.

freeze Hibernation-specific, executed before creating a hibernation image. Analogous to **suspend()**, but it should not enable the device to signal wakeup events or change its power state. The majority of subsystems (with the notable exception of the PCI bus type) expect the driver-level **freeze()** to save the device settings in memory to be used by **restore()** during the subsequent

resume from hibernation. Subsystem-level **freeze()** is executed for all devices after invoking subsystem-level **prepare()** for all of them.

thaw Hibernation-specific, executed after creating a hibernation image OR if the creation of an image has failed. Also executed after a failing attempt to restore the contents of main memory from such an image. Undo the changes made by the preceding **freeze()**, so the device can be operated in the same way as immediately before the call to **freeze()**. Subsystem-level **thaw()** is executed for all devices after invoking subsystem-level **thaw_noirq()** for all of them. It also may be executed directly after **freeze()** in case of a transition error.

poweroff Hibernation-specific, executed after saving a hibernation image. Analogous to **suspend()**, but it need not save the device's settings in memory. Subsystem-level **poweroff()** is executed for all devices after invoking subsystem-level **prepare()** for all of them.

restore Hibernation-specific, executed after restoring the contents of main memory from a hibernation image, analogous to **resume()**.

suspend_late Continue operations started by **suspend()**. For a number of devices **suspend_late()** may point to the same callback routine as the runtime suspend callback.

resume_early Prepare to execute **resume()**. For a number of devices **resume_early()** may point to the same callback routine as the runtime resume callback.

freeze_late Continue operations started by **freeze()**. Analogous to **suspend_late()**, but it should not enable the device to signal wakeup events or change its power state.

thaw_early Prepare to execute **thaw()**. Undo the changes made by the preceding **freeze_late()**.

poweroff_late Continue operations started by **poweroff()**. Analogous to **suspend_late()**, but it need not save the device's settings in memory.

restore_early Prepare to execute **restore()**, analogous to **resume_early()**.

suspend_noirq Complete the actions started by **suspend()**. Carry out any additional operations required for suspending the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **suspend_noirq()** is being executed. It generally is expected that the device will be in a low-power state (appropriate for the target system sleep state) after subsystem-level **suspend_noirq()** has returned successfully. If the device can generate system wakeup signals and is enabled to wake up the system, it should be configured to do so at that time. However, depending on the platform and device's subsystem, **suspend()** or **suspend_late()** may be allowed to put the device into the low-power state and configure it to generate wakeup signals, in which case it generally is not necessary to define **suspend_noirq()**.

resume_noirq Prepare for the execution of **resume()** by carrying out any operations required for resuming the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **resume_noirq()** is being executed.

freeze_noirq Complete the actions started by **freeze()**. Carry out any additional operations required for freezing the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **freeze_noirq()** is being executed. The power state of the device should not be changed by either **freeze()**, or **freeze_late()**, or **freeze_noirq()** and it should not be configured to signal system wakeup by any of these callbacks.

thaw_noirq Prepare for the execution of **thaw()** by carrying out any operations required for thawing the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **thaw_noirq()** is being executed.

poweroff_noirq Complete the actions started by **poweroff()**. Analogous to **suspend_noirq()**, but it need not save the device's settings in memory.

restore_noirq Prepare for the execution of **restore()** by carrying out any operations required for thawing the device that might be racing with its driver's interrupt handler, which is guaranteed not to run while **restore_noirq()** is being executed. Analogous to **resume_noirq()**.

runtime_suspend Prepare the device for a condition in which it won't be able to communicate with the CPU(s) and RAM due to power management. This need not mean that the device should be put into a low-power state. For example, if the device is behind a link which is about to be turned off, the device may remain at full power. If the device does go to low power and is capable of generating runtime wakeup events, remote wakeup (i.e., a hardware mechanism allowing the device to request a change of its power state via an interrupt) should be enabled for it.

runtime_resume Put the device into the fully active state in response to a wakeup event generated by hardware or at the request of software. If necessary, put the device into the full-power state and restore its registers, so that it is fully operational.

runtime_idle Device appears to be inactive and it might be put into a low-power state if all of the necessary conditions are satisfied. Check these conditions, and return 0 if it's appropriate to let the PM core queue a suspend request for the device.

Description

Several device power state transitions are externally visible, affecting the state of pending I/O queues and (for drivers that touch hardware) interrupts, wakeups, DMA, and other hardware state. There may also be internal transitions to various low-power modes which are transparent to the rest of the driver stack (such as a driver that's ON gating off clocks which are not in active use).

The externally visible transitions are handled with the help of callbacks included in this structure in such a way that, typically, two levels of callbacks are involved. First, the PM core executes callbacks provided by PM domains, device types, classes and bus types. They are the subsystem-level callbacks expected to execute callbacks provided by device drivers, although they may choose not to do that. If the driver callbacks are executed, they have to collaborate with the subsystem-level callbacks to achieve the goals appropriate for the given system transition, given transition phase and the subsystem the device belongs to.

All of the above callbacks, except for **complete()**, return error codes. However, the error codes returned by **resume()**, **thaw()**, **restore()**, **resume_noirq()**, **thaw_noirq()**, and **restore_noirq()**, do not cause the PM core to abort the resume transition during which they are returned. The error codes returned in those cases are only printed to the system logs for debugging purposes. Still, it is recommended that drivers only return error codes from their resume methods in case of an unrecoverable failure (i.e. when the device being handled refuses to resume and becomes unusable) to allow the PM core to be modified in the future, so that it can avoid attempting to handle devices that failed to resume and their children.

It is allowed to unregister devices while the above callbacks are being executed. However, a callback routine MUST NOT try to unregister the device it was called for, although it may unregister children of that device (for example, if it detects that a child was unplugged while the system was asleep).

There also are callbacks related to runtime power management of devices. Again, as a rule these callbacks are executed by the PM core for subsystems (PM domains, device types, classes and bus types) and the subsystem-level callbacks are expected to invoke the driver callbacks. Moreover, the exact actions to be performed by a device driver's callbacks generally depend on the platform and subsystem the device belongs to.

Refer to Documentation/power/runtime_pm.rst for more information about the role of the **runtime_suspend()**, **runtime_resume()** and **runtime_idle()** callbacks in device runtime power management.

struct **dev_pm_domain**
power management domain representation.

Definition

```
struct dev_pm_domain {
  struct dev_pm_ops       ops;
  int (*start)(struct device *dev);
  void (*detach)(struct device *dev, bool power_off);
  int (*activate)(struct device *dev);
  void (*sync)(struct device *dev);
  void (*dismiss)(struct device *dev);
};
```

Members

ops Power management operations associated with this domain.

start Called when a user needs to start the device via the domain.

detach Called when removing a device from the domain.

activate Called before executing probe routines for bus types and drivers.

sync Called after successful driver probe.

dismiss Called after unsuccessful driver probe and after driver removal.

Description

Power domains provide callbacks that are executed during system suspend, hibernation, system resume and during runtime PM transitions instead of subsystem-level and driver-level callbacks.

CHAPTER

SEVEN

THE COMMON CLK FRAMEWORK

Author Mike Turquette <mturquette@ti.com>

This document endeavours to explain the common clk framework details, and how to port a platform over to this framework. It is not yet a detailed explanation of the clock api in include/linux/clk.h, but perhaps someday it will include that information.

7.1 Introduction and interface split

The common clk framework is an interface to control the clock nodes available on various devices today. This may come in the form of clock gating, rate adjustment, muxing or other operations. This framework is enabled with the CONFIG_COMMON_CLK option.

The interface itself is divided into two halves, each shielded from the details of its counterpart. First is the common definition of struct clk which unifies the framework-level accounting and infrastructure that has traditionally been duplicated across a variety of platforms. Second is a common implementation of the clk.h api, defined in drivers/clk/clk.c. Finally there is struct clk_ops, whose operations are invoked by the clk api implementation.

The second half of the interface is comprised of the hardware-specific callbacks registered with struct clk_ops and the corresponding hardware-specific structures needed to model a particular clock. For the remainder of this document any reference to a callback in struct clk_ops, such as .enable or .set_rate, implies the hardware-specific implementation of that code. Likewise, references to struct clk_foo serve as a convenient shorthand for the implementation of the hardware-specific bits for the hypothetical "foo" hardware.

Tying the two halves of this interface together is struct clk_hw, which is defined in struct clk_foo and pointed to within struct clk_core. This allows for easy navigation between the two discrete halves of the common clock interface.

7.2 Common data structures and api

Below is the common struct clk_core definition from drivers/clk/clk.c, modified for brevity:

```
struct clk_core {
        const char              *name;
        const struct clk_ops    *ops;
        struct clk_hw           *hw;
        struct module           *owner;
        struct clk_core         *parent;
        const char              **parent_names;
        struct clk_core         **parents;
        u8                      num_parents;
        u8                      new_parent_index;
        ...
};
```

The members above make up the core of the clk tree topology. The clk api itself defines several driver-facing functions which operate on struct clk. That api is documented in include/linux/clk.h.

Platforms and devices utilizing the common struct clk_core use the struct clk_ops pointer in struct clk_core to perform the hardware-specific parts of the operations defined in clk-provider.h:

```
struct clk_ops {
        int             (*prepare)(struct clk_hw *hw);
        void            (*unprepare)(struct clk_hw *hw);
        int             (*is_prepared)(struct clk_hw *hw);
        void            (*unprepare_unused)(struct clk_hw *hw);
        int             (*enable)(struct clk_hw *hw);
        void            (*disable)(struct clk_hw *hw);
        int             (*is_enabled)(struct clk_hw *hw);
        void            (*disable_unused)(struct clk_hw *hw);
        unsigned long   (*recalc_rate)(struct clk_hw *hw,
                                       unsigned long parent_rate);
        long            (*round_rate)(struct clk_hw *hw,
                                      unsigned long rate,
                                      unsigned long *parent_rate);
        int             (*determine_rate)(struct clk_hw *hw,
                                          struct clk_rate_request *req);
        int             (*set_parent)(struct clk_hw *hw, u8 index);
        u8              (*get_parent)(struct clk_hw *hw);
        int             (*set_rate)(struct clk_hw *hw,
                                    unsigned long rate,
                                    unsigned long parent_rate);
        int             (*set_rate_and_parent)(struct clk_hw *hw,
                                    unsigned long rate,
                                    unsigned long parent_rate,
                                    u8 index);
        unsigned long   (*recalc_accuracy)(struct clk_hw *hw,
                                       unsigned long parent_accuracy);
        int             (*get_phase)(struct clk_hw *hw);
        int             (*set_phase)(struct clk_hw *hw, int degrees);
        void            (*init)(struct clk_hw *hw);
```

(continues on next page)

(continued from previous page)

```
        void            (*debug_init)(struct clk_hw *hw,
                                      struct dentry *dentry);
};
```

7.3 Hardware clk implementations

The strength of the common struct clk_core comes from its .ops and .hw pointers which abstract the details of struct clk from the hardware-specific bits, and vice versa. To illustrate consider the simple gateable clk implementation in drivers/clk/clk-gate.c:

```
struct clk_gate {
        struct clk_hw   hw;
        void __iomem    *reg;
        u8              bit_idx;
        ...
};
```

struct clk_gate contains struct clk_hw hw as well as hardware-specific knowledge about which register and bit controls this clk's gating. Nothing about clock topology or accounting, such as enable_count or notifier_count, is needed here. That is all handled by the common framework code and struct clk_core.

Let's walk through enabling this clk from driver code:

```
struct clk *clk;
clk = clk_get(NULL, "my_gateable_clk");

clk_prepare(clk);
clk_enable(clk);
```

The call graph for clk_enable is very simple:

```
clk_enable(clk);
        clk->ops->enable(clk->hw);
        [resolves to...]
                clk_gate_enable(hw);
                [resolves struct clk gate with to_clk_gate(hw)]
                        clk_gate_set_bit(gate);
```

And the definition of clk_gate_set_bit:

```
static void clk_gate_set_bit(struct clk_gate *gate)
{
        u32 reg;

        reg = __raw_readl(gate->reg);
        reg |= BIT(gate->bit_idx);
        writel(reg, gate->reg);
}
```

Note that to_clk_gate is defined as:

```
#define to_clk_gate(_hw) container_of(_hw, struct clk_gate, hw)
```

This pattern of abstraction is used for every clock hardware representation.

7.4 Supporting your own clk hardware

When implementing support for a new type of clock it is only necessary to include the following header:

```
#include <linux/clk-provider.h>
```

To construct a clk hardware structure for your platform you must define the following:

```
struct clk_foo {
        struct clk_hw hw;
        ... hardware specific data goes here ...
};
```

To take advantage of your data you'll need to support valid operations for your clk:

```
struct clk_ops clk_foo_ops = {
        .enable         = &clk_foo_enable,
        .disable        = &clk_foo_disable,
};
```

Implement the above functions using container_of:

```
#define to_clk_foo(_hw) container_of(_hw, struct clk_foo, hw)

int clk_foo_enable(struct clk_hw *hw)
{
        struct clk_foo *foo;

        foo = to_clk_foo(hw);

        ... perform magic on foo ...

        return 0;
};
```

Below is a matrix detailing which clk_ops are mandatory based upon the hardware capabilities of that clock. A cell marked as "y" means mandatory, a cell marked as "n" implies that either including that callback is invalid or otherwise unnecessary. Empty cells are either optional or must be evaluated on a case-by-case basis.

Table 1: clock hardware characteristics

	gate	change rate	single parent	multiplexer	root
.prepare					
.unprepare					
.enable	y				
.disable	y				
.is_enabled	y				
.recalc_rate		y			
.round_rate		y[1]			
.determine_rate		y[1]			
.set_rate		y			
.set_parent			n	y	n
.get_parent			n	y	n
.recalc_accuracy					
.init					

Finally, register your clock at run-time with a hardware-specific registration function. This function simply populates struct clk_foo's data and then passes the common struct clk parameters to the framework with a call to:

```
clk_register(...)
```

See the basic clock types in `drivers/clk/clk-*.c` for examples.

7.5 Disabling clock gating of unused clocks

Sometimes during development it can be useful to be able to bypass the default disabling of unused clocks. For example, if drivers aren't enabling clocks properly but rely on them being on from the bootloader, bypassing the disabling means that the driver will remain functional while the issues are sorted out.

To bypass this disabling, include "clk_ignore_unused" in the bootargs to the kernel.

[1] either one of round_rate or determine_rate is required.

7.6 Locking

The common clock framework uses two global locks, the prepare lock and the enable lock.

The enable lock is a spinlock and is held across calls to the .enable, .disable operations. Those operations are thus not allowed to sleep, and calls to the clk_enable(), clk_disable() API functions are allowed in atomic context.

For clk_is_enabled() API, it is also designed to be allowed to be used in atomic context. However, it doesn't really make any sense to hold the enable lock in core, unless you want to do something else with the information of the enable state with that lock held. Otherwise, seeing if a clk is enabled is a one-shot read of the enabled state, which could just as easily change after the function returns because the lock is released. Thus the user of this API needs to handle synchronizing the read of the state with whatever they're using it for to make sure that the enable state doesn't change during that time.

The prepare lock is a mutex and is held across calls to all other operations. All those operations are allowed to sleep, and calls to the corresponding API functions are not allowed in atomic context.

This effectively divides operations in two groups from a locking perspective.

Drivers don't need to manually protect resources shared between the operations of one group, regardless of whether those resources are shared by multiple clocks or not. However, access to resources that are shared between operations of the two groups needs to be protected by the drivers. An example of such a resource would be a register that controls both the clock rate and the clock enable/disable state.

The clock framework is reentrant, in that a driver is allowed to call clock framework functions from within its implementation of clock operations. This can for instance cause a .set_rate operation of one clock being called from within the .set_rate operation of another clock. This case must be considered in the driver implementations, but the code flow is usually controlled by the driver in that case.

Note that locking must also be considered when code outside of the common clock framework needs to access resources used by the clock operations. This is considered out of scope of this document.

CHAPTER

EIGHT

BUS-INDEPENDENT DEVICE ACCESSES

Author Matthew Wilcox

Author Alan Cox

8.1 Introduction

Linux provides an API which abstracts performing IO across all busses and devices, allowing device drivers to be written independently of bus type.

8.2 Memory Mapped IO

8.2.1 Getting Access to the Device

The most widely supported form of IO is memory mapped IO. That is, a part of the CPU's address space is interpreted not as accesses to memory, but as accesses to a device. Some architectures define devices to be at a fixed address, but most have some method of discovering devices. The PCI bus walk is a good example of such a scheme. This document does not cover how to receive such an address, but assumes you are starting with one. Physical addresses are of type unsigned long.

This address should not be used directly. Instead, to get an address suitable for passing to the accessor functions described below, you should call `ioremap()`. An address suitable for accessing the device will be returned to you.

After you've finished using the device (say, in your module's exit routine), call `iounmap()` in order to return the address space to the kernel. Most architectures allocate new address space each time you call `ioremap()`, and they can run out unless you call `iounmap()`.

8.2.2 Accessing the device

The part of the interface most used by drivers is reading and writing memory-mapped registers on the device. Linux provides interfaces to read and write 8-bit, 16-bit, 32-bit and 64-bit quantities. Due to a historical accident, these are named byte, word, long and quad accesses. Both read and write accesses are supported; there is no prefetch support at this time.

The functions are named readb(), readw(), readl(), readq(), readb_relaxed(), readw_relaxed(), readl_relaxed(), readq_relaxed(), writeb(), writew(), writel() and writeq().

Some devices (such as framebuffers) would like to use larger transfers than 8 bytes at a time. For these devices, the `memcpy_toio()`, `memcpy_fromio()` and `memset_io()` functions are provided. Do not use memset or memcpy on IO addresses; they are not guaranteed to copy data in order.

The read and write functions are defined to be ordered. That is the compiler is not permitted to reorder the I/O sequence. When the ordering can be compiler optimised, you can use __readb() and friends to indicate the relaxed ordering. Use this with care.

While the basic functions are defined to be synchronous with respect to each other and ordered with respect to each other the busses the devices sit on may themselves have asynchronicity. In particular many authors are burned by the fact that PCI bus writes are posted asynchronously. A driver author must issue a read from the same device to ensure that writes have occurred in the specific cases the author cares. This kind of property cannot be hidden from driver writers in the API. In some cases, the read used to flush the device may be expected to fail (if the card is resetting, for example). In that case, the read should be done from config space, which is guaranteed to soft-fail if the card doesn't respond.

The following is an example of flushing a write to a device when the driver would like to ensure the write's effects are visible prior to continuing execution:

```
static inline void
qla1280_disable_intrs(struct scsi_qla_host *ha)
{
    struct device_reg *reg;

    reg = ha->iobase;
    /* disable risc and host interrupts */
    WRT_REG_WORD(&reg->ictrl, 0);
    /*
     * The following read will ensure that the above write
     * has been received by the device before we return from this
     * function.
     */
    RD_REG_WORD(&reg->ictrl);
    ha->flags.ints_enabled = 0;
}
```

PCI ordering rules also guarantee that PIO read responses arrive after any outstanding DMA writes from that bus, since for some devices the result of a readb() call may signal to the driver that a DMA transaction is complete. In many cases, however, the driver may want to indicate that the next readb() call has no rela-

tion to any previous DMA writes performed by the device. The driver can use readb_relaxed() for these cases, although only some platforms will honor the relaxed semantics. Using the relaxed read functions will provide significant performance benefits on platforms that support it. The qla2xxx driver provides examples of how to use readX_relaxed(). In many cases, a majority of the driver's readX() calls can safely be converted to readX_relaxed() calls, since only a few will indicate or depend on DMA completion.

8.3 Port Space Accesses

8.3.1 Port Space Explained

Another form of IO commonly supported is Port Space. This is a range of addresses separate to the normal memory address space. Access to these addresses is generally not as fast as accesses to the memory mapped addresses, and it also has a potentially smaller address space.

Unlike memory mapped IO, no preparation is required to access port space.

8.3.2 Accessing Port Space

Accesses to this space are provided through a set of functions which allow 8-bit, 16-bit and 32-bit accesses; also known as byte, word and long. These functions are `inb()`, `inw()`, `inl()`, `outb()`, `outw()` and `outl()`.

Some variants are provided for these functions. Some devices require that accesses to their ports are slowed down. This functionality is provided by appending a `_p` to the end of the function. There are also equivalents to memcpy. The `ins()` and `outs()` functions copy bytes, words or longs to the given port.

8.4 Public Functions Provided

phys_addr_t **virt_to_phys**(volatile void * address)
map virtual addresses to physical

Parameters

volatile void * address address to remap

The returned physical address is the physical (CPU) mapping for the memory address given. It is only valid to use this function on addresses directly mapped or allocated via kmalloc.

This function does not give bus mappings for DMA transfers. In almost all conceivable cases a device driver should not be using this function

void * **phys_to_virt**(phys_addr_t address)
map physical address to virtual

Parameters

phys_addr_t address address to remap

The returned virtual address is a current CPU mapping for the memory address given. It is only valid to use this function on addresses that have a kernel mapping

This function does not handle bus mappings for DMA transfers. In almost all conceivable cases a device driver should not be using this function

void __iomem * **ioremap**(resource_size_t offset, unsigned long size)
map bus memory into CPU space

Parameters

resource_size_t offset bus address of the memory

unsigned long size size of the resource to map

Description

ioremap performs a platform specific sequence of operations to make bus memory CPU accessible via the readb/readw/readl/writeb/ writew/writel functions and the other mmio helpers. The returned address is not guaranteed to be usable directly as a virtual address.

If the area you are trying to map is a PCI BAR you should have a look at `pci_iomap()`.

void **iosubmit_cmds512**(void __iomem * __dst, const void * src, size_t count)
copy data to single MMIO location, in 512-bit units

Parameters

void __iomem * __dst destination, in MMIO space (must be 512-bit aligned)

const void * src source

size_t count number of 512 bits quantities to submit

Description

Submit data from kernel space to MMIO space, in units of 512 bits at a time. Order of access is not guaranteed, nor is a memory barrier performed afterwards.

Warning: Do not use this helper unless your driver has checked that the CPU instruction is supported on the platform.

void __iomem * **pci_iomap_range**(struct pci_dev * dev, int bar, unsigned long offset, unsigned long maxlen)
create a virtual mapping cookie for a PCI BAR

Parameters

struct pci_dev * dev PCI device that owns the BAR

int bar BAR number

unsigned long offset map memory at the given offset in BAR

unsigned long maxlen max length of the memory to map

Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way.

maxlen specifies the maximum length to map. If you want to get access to the complete BAR from offset to the end, pass `0` here.

void __iomem * **pci_iomap_wc_range**(struct pci_dev * dev, int bar, unsigned long offset, unsigned long maxlen)
create a virtual WC mapping cookie for a PCI BAR

Parameters

struct pci_dev * dev PCI device that owns the BAR

int bar BAR number

unsigned long offset map memory at the given offset in BAR

unsigned long maxlen max length of the memory to map

Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way. When possible write combining is used.

maxlen specifies the maximum length to map. If you want to get access to the complete BAR from offset to the end, pass `0` here.

void __iomem * **pci_iomap**(struct pci_dev * dev, int bar, unsigned long maxlen)
create a virtual mapping cookie for a PCI BAR

Parameters

struct pci_dev * dev PCI device that owns the BAR

int bar BAR number

unsigned long maxlen length of the memory to map

Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way.

maxlen specifies the maximum length to map. If you want to get access to the complete BAR without checking for its length first, pass `0` here.

void __iomem * **pci_iomap_wc**(struct pci_dev * dev, int bar, unsigned long maxlen)
create a virtual WC mapping cookie for a PCI BAR

Parameters

struct pci_dev * dev PCI device that owns the BAR

int bar BAR number

unsigned long maxlen length of the memory to map

Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way. When possible write combining is used.

maxlen specifies the maximum length to map. If you want to get access to the complete BAR without checking for its length first, pass `0` here.

CHAPTER

NINE

DEVICE CONNECTIONS

9.1 Introduction

Devices often have connections to other devices that are outside of the direct child/parent relationship. A serial or network communication controller, which could be a PCI device, may need to be able to get a reference to its PHY component, which could be attached for example to the I2C bus. Some device drivers need to be able to control the clocks or the GPIOs for their devices, and so on.

Device connections are generic descriptions of any type of connection between two separate devices.

Device connections alone do not create a dependency between the two devices. They are only descriptions which are not tied to either of the devices directly. A dependency between the two devices exists only if one of the two endpoint devices requests a reference to the other. The descriptions themselves can be defined in firmware (not yet supported) or they can be built-in.

9.2 Usage

Device connections should exist before device `->probe` callback is called for either endpoint device in the description. If the connections are defined in firmware, this is not a problem. It should be considered if the connection descriptions are "built-in", and need to be added separately.

The connection description consists of the names of the two devices with the connection, i.e. the endpoints, and unique identifier for the connection which is needed if there are multiple connections between the two devices.

After a description exists, the devices in it can request reference to the other endpoint device, or they can request the description itself.

9.3 API

void * **device_connection_find_match**(struct device * dev, const char * con_id, void * data, devcon_match_fn_t match)
Find physical connection to a device

Parameters

struct device * dev Device with the connection

const char * con_id Identifier for the connection

void * data Data for the match function

devcon_match_fn_t match Function to check and convert the connection description

Description

Find a connection with unique identifier **con_id** between **dev** and another device. **match** will be used to convert the connection description to data the caller is expecting to be returned.

struct device * **device_connection_find**(struct device * dev, const char * con_id)
Find two devices connected together

Parameters

struct device * dev Device with the connection

const char * con_id Identifier for the connection

Description

Find a connection with unique identifier **con_id** between **dev** and another device. On success returns handle to the device that is connected to **dev**, with the reference count for the found device incremented. Returns NULL if no matching connection was found, or ERR_PTR(-EPROBE_DEFER) when a connection was found but the other device has not been enumerated yet.

void **device_connection_add**(struct device_connection * con)
Register a connection description

Parameters

struct device_connection * con The connection description to be registered

void **device_connection_remove**(struct device_connection * con)
Unregister connection description

Parameters

struct device_connection * con The connection description to be unregistered

CHAPTER

TEN

BUFFER SHARING AND SYNCHRONIZATION

The dma-buf subsystem provides the framework for sharing buffers for hardware (DMA) access across multiple device drivers and subsystems, and for synchronizing asynchronous hardware access.

This is used, for example, by drm “prime” multi-GPU support, but is of course not limited to GPU use cases.

The three main components of this are: (1) dma-buf, representing a sg_table and exposed to userspace as a file descriptor to allow passing between devices, (2) fence, which provides a mechanism to signal when one device has finished access, and (3) reservation, which manages the shared or exclusive fence(s) associated with the buffer.

10.1 Shared DMA Buffers

This document serves as a guide to device-driver writers on what is the dma-buf buffer sharing API, how to use it for exporting and using shared buffers.

Any device driver which wishes to be a part of DMA buffer sharing, can do so as either the ‘exporter’ of buffers, or the ‘user’ or ‘importer’ of buffers.

Say a driver A wants to use buffers created by driver B, then we call B as the exporter, and A as buffer-user/importer.

The exporter

- implements and manages operations in `struct dma_buf_ops` for the buffer,
- allows other users to share the buffer by using dma_buf sharing APIs,
- manages the details of buffer allocation, wrapped in a `struct dma_buf`,
- decides about the actual backing storage where this allocation happens,
- and takes care of any migration of scatterlist - for all (shared) users of this buffer.

The buffer-user

- is one of (many) sharing users of the buffer.
- doesn’t need to worry about how the buffer is allocated, or where.

- and needs a mechanism to get access to the scatterlist that makes up this buffer in memory, mapped into its own address space, so it can access the same area of memory. This interface is provided by `struct dma_buf_attachment`.

Any exporters or users of the dma-buf buffer sharing framework must have a 'select DMA_SHARED_BUFFER' in their respective Kconfigs.

10.1.1 Userspace Interface Notes

Mostly a DMA buffer file descriptor is simply an opaque object for userspace, and hence the generic interface exposed is very minimal. There's a few things to consider though:

- Since kernel 3.12 the dma-buf FD supports the llseek system call, but only with offset=0 and whence=SEEK_END|SEEK_SET. SEEK_SET is supported to allow the usual size discover pattern size = SEEK_END(0); SEEK_SET(0). Every other llseek operation will report -EINVAL.

  If llseek on dma-buf FDs isn't support the kernel will report -ESPIPE for all cases. Userspace can use this to detect support for discovering the dma-buf size using llseek.

- In order to avoid fd leaks on exec, the FD_CLOEXEC flag must be set on the file descriptor. This is not just a resource leak, but a potential security hole. It could give the newly exec'd application access to buffers, via the leaked fd, to which it should otherwise not be permitted access.

  The problem with doing this via a separate fcntl() call, versus doing it atomically when the fd is created, is that this is inherently racy in a multi-threaded app[3]. The issue is made worse when it is library code opening/creating the file descriptor, as the application may not even be aware of the fd's.

  To avoid this problem, userspace must have a way to request O_CLOEXEC flag be set when the dma-buf fd is created. So any API provided by the exporting driver to create a dmabuf fd must provide a way to let userspace control setting of O_CLOEXEC flag passed in to `dma_buf_fd()`.

- Memory mapping the contents of the DMA buffer is also supported. See the discussion below on CPU Access to DMA Buffer Objects for the full details.

- The DMA buffer FD is also pollable, see Fence Poll Support below for details.

10.1.2 Basic Operation and Device DMA Access

For device DMA access to a shared DMA buffer the usual sequence of operations is fairly simple:

1. The exporter defines his exporter instance using `DEFINE_DMA_BUF_EXPORT_INFO()` and calls `dma_buf_export()` to wrap a private buffer object into a `dma_buf`. It then exports that `dma_buf` to userspace as a file descriptor by calling `dma_buf_fd()`.

2. Userspace passes this file-descriptors to all drivers it wants this buffer to share with: First the filedescriptor is converted to a `dma_buf` using `dma_buf_get()`. Then the buffer is attached to the device using `dma_buf_attach()`.

   Up to this stage the exporter is still free to migrate or reallocate the backing storage.

3. Once the buffer is attached to all devices userspace can initiate DMA access to the shared buffer. In the kernel this is done by calling `dma_buf_map_attachment()` and `dma_buf_unmap_attachment()`.

4. Once a driver is done with a shared buffer it needs to call `dma_buf_detach()` (after cleaning up any mappings) and then release the reference acquired with dma_buf_get by calling `dma_buf_put()`.

For the detailed semantics exporters are expected to implement see `dma_buf_ops`.

10.1.3 CPU Access to DMA Buffer Objects

There are mutliple reasons for supporting CPU access to a dma buffer object:

- Fallback operations in the kernel, for example when a device is connected over USB and the kernel needs to shuffle the data around first before sending it away. Cache coherency is handled by braketing any transactions with calls to `dma_buf_begin_cpu_access()` and `dma_buf_end_cpu_access()` access.

  Since for most kernel internal dma-buf accesses need the entire buffer, a vmap interface is introduced. Note that on very old 32-bit architectures vmalloc space might be limited and result in vmap calls failing.

  Interfaces:: void *dma_buf_vmap(struct dma_buf *dmabuf) void dma_buf_vunmap(struct dma_buf *dmabuf, void *vaddr)

  The vmap call can fail if there is no vmap support in the exporter, or if it runs out of vmalloc space. Fallback to kmap should be implemented. Note that the dma-buf layer keeps a reference count for all vmap access and calls down into the exporter's vmap function only when no vmapping exists, and only unmaps it once. Protection against concurrent vmap/vunmap calls is provided by taking the dma_buf->lock mutex.

- For full compatibility on the importer side with existing userspace interfaces, which might already support mmap'ing buffers. This is needed in many processing pipelines (e.g. feeding a software rendered image into a hardware pipeline, thumbnail creation, snapshots, ⋯). Also, Android's ION framework already supported this and for DMA buffer file descriptors to replace ION buffers mmap support was needed.

  There is no special interfaces, userspace simply calls mmap on the dma-buf fd. But like for CPU access there's a need to braket the actual access, which is handled by the ioctl (DMA_BUF_IOCTL_SYNC). Note that DMA_BUF_IOCTL_SYNC can fail with -EAGAIN or -EINTR, in which case it must be restarted.

  Some systems might need some sort of cache coherency management e.g. when CPU and GPU domains are being accessed through dma-buf

at the same time. To circumvent this problem there are begin/end coherency markers, that forward directly to existing dma-buf device drivers vfunc hooks. Userspace can make use of those markers through the DMA_BUF_IOCTL_SYNC ioctl. The sequence would be used like following:

- mmap dma-buf fd
- for each drawing/upload cycle in CPU 1. SYNC_START ioctl, 2. read/write to mmap area 3. SYNC_END ioctl. This can be repeated as often as you want (with the new data being consumed by say the GPU or the scanout device)
- munmap once you don't need the buffer any more

For correctness and optimal performance, it is always required to use SYNC_START and SYNC_END before and after, respectively, when accessing the mapped address. Userspace cannot rely on coherent access, even when there are systems where it just works without calling these ioctls.

- And as a CPU fallback in userspace processing pipelines.

  Similar to the motivation for kernel cpu access it is again important that the userspace code of a given importing subsystem can use the same interfaces with a imported dma-buf buffer object as with a native buffer object. This is especially important for drm where the userspace part of contemporary OpenGL, X, and other drivers is huge, and reworking them to use a different way to mmap a buffer rather invasive.

  The assumption in the current dma-buf interfaces is that redirecting the initial mmap is all that's needed. A survey of some of the existing subsystems shows that no driver seems to do any nefarious thing like syncing up with outstanding asynchronous processing on the device or allocating special resources at fault time. So hopefully this is good enough, since adding interfaces to intercept pagefaults and allow pte shootdowns would increase the complexity quite a bit.

  Interface::

  int dma_buf_mmap(struct dma_buf *, struct vm_area_struct *, unsigned long);

  If the importing subsystem simply provides a special-purpose mmap call to set up a mapping in userspace, calling do_mmap with dma_buf->file will equally achieve that for a dma-buf object.

10.1.4 Fence Poll Support

To support cross-device and cross-driver synchronization of buffer access implicit fences (represented internally in the kernel with `struct fence`) can be attached to a `dma_buf`. The glue for that and a few related things are provided in the `dma_resv` structure.

Userspace can query the state of these implicitly tracked fences using poll() and related system calls:

- Checking for EPOLLIN, i.e. read access, can be use to query the state of the most recent write or exclusive fence.
- Checking for EPOLLOUT, i.e. write access, can be used to query the state of all attached fences, shared and exclusive ones.

Note that this only signals the completion of the respective fences, i.e. the DMA transfers are complete. Cache flushing and any other necessary preparations before CPU access can begin still need to happen.

10.1.5 Kernel Functions and Structures Reference

struct dma_buf * **dma_buf_export**(const struct dma_buf_export_info * exp_info)
Creates a new dma_buf, and associates an anon file with this buffer, so it can be exported. Also connect the allocator specific data and ops to the buffer. Additionally, provide a name string for exporter; useful in debugging.

Parameters

const struct dma_buf_export_info * exp_info [in] holds all the export related information provided by the exporter. see `struct dma_buf_export_info` for further details.

Description

Returns, on success, a newly created dma_buf object, which wraps the supplied private data and operations for dma_buf_ops. On either missing ops, or error in allocating struct dma_buf, will return negative error.

For most cases the easiest way to create **exp_info** is through the `DEFINE_DMA_BUF_EXPORT_INFO` macro.

int **dma_buf_fd**(struct dma_buf * dmabuf, int flags)
returns a file descriptor for the given dma_buf

Parameters

struct dma_buf * dmabuf [in] pointer to dma_buf for which fd is required.

int flags [in] flags to give to fd

Description

On success, returns an associated ‘fd’. Else, returns error.

struct dma_buf * **dma_buf_get**(int fd)
returns the dma_buf structure related to an fd

Parameters

int fd [in] fd associated with the dma_buf to be returned

Description

On success, returns the dma_buf structure associated with an fd; uses file’s refcounting done by fget to increase refcount. returns ERR_PTR otherwise.

void **dma_buf_put**(struct dma_buf * dmabuf)
decreases refcount of the buffer

Parameters

struct dma_buf * dmabuf [in] buffer to reduce refcount of

Description

Uses file's refcounting done implicitly by fput().

If, as a result of this call, the refcount becomes 0, the 'release' file operation related to this fd is called. It calls `dma_buf_ops.release` vfunc in turn, and frees the memory allocated for dmabuf when exported.

struct dma_buf_attachment * **dma_buf_dynamic_attach**(struct dma_buf * dmabuf, struct device * dev, const struct dma_buf_attach_ops * importer_ops, void * importer_priv)

Add the device to dma_buf's attachments list; optionally, calls attach() of dma_buf_ops to allow device-specific attach functionality

Parameters

struct dma_buf * dmabuf [in] buffer to attach device to.

struct device * dev [in] device to be attached.

const struct dma_buf_attach_ops * importer_ops [in] importer operations for the attachment

void * importer_priv [in] importer private pointer for the attachment

Description

Returns struct dma_buf_attachment pointer for this attachment. Attachments must be cleaned up by calling `dma_buf_detach()`.

A pointer to newly created `dma_buf_attachment` on success, or a negative error code wrapped into a pointer on failure.

Note that this can fail if the backing storage of **dmabuf** is in a place not accessible to **dev**, and cannot be moved to a more suitable place. This is indicated with the error code -EBUSY.

Return

struct dma_buf_attachment * **dma_buf_attach**(struct dma_buf * dmabuf, struct device * dev)

Wrapper for dma_buf_dynamic_attach

Parameters

struct dma_buf * dmabuf [in] buffer to attach device to.

struct device * dev [in] device to be attached.

Description

Wrapper to call `dma_buf_dynamic_attach()` for drivers which still use a static mapping.

void **dma_buf_detach**(struct dma_buf * dmabuf, struct dma_buf_attachment * attach)
Remove the given attachment from dmabuf's attachments list; optionally calls detach() of dma_buf_ops for device-specific detach

Parameters

struct dma_buf * dmabuf [in] buffer to detach from.

struct dma_buf_attachment * attach [in] attachment to be detached; is free'd after this call.

Description

Clean up a device attachment obtained by calling `dma_buf_attach()`.

int **dma_buf_pin**(struct dma_buf_attachment * attach)
Lock down the DMA-buf

Parameters

struct dma_buf_attachment * attach [in] attachment which should be pinned

Return

0 on success, negative error code on failure.

void **dma_buf_unpin**(struct dma_buf_attachment * attach)
Remove lock from DMA-buf

Parameters

struct dma_buf_attachment * attach [in] attachment which should be unpinned

struct sg_table * **dma_buf_map_attachment**(struct dma_buf_attachment * attach, enum dma_data_direction direction)
Returns the scatterlist table of the attachment; mapped into _device_ address space. Is a wrapper for map_dma_buf() of the dma_buf_ops.

Parameters

struct dma_buf_attachment * attach [in] attachment whose scatterlist is to be returned

enum dma_data_direction direction [in] direction of DMA transfer

Description

Returns sg_table containing the scatterlist to be returned; returns ERR_PTR on error. May return -EINTR if it is interrupted by a signal.

A mapping must be unmapped by using `dma_buf_unmap_attachment()`. Note that the underlying backing storage is pinned for as long as a mapping exists, therefore users/importers should not hold onto a mapping for undue amounts of time.

void **dma_buf_unmap_attachment**(struct dma_buf_attachment * attach, struct sg_table * sg_table, enum dma_data_direction direction)
unmaps and decreases usecount of the buffer;might deallocate the scatterlist associated. Is a wrapper for unmap_dma_buf() of dma_buf_ops.

Parameters

struct dma_buf_attachment * attach [in] attachment to unmap buffer from

struct sg_table * sg_table [in] scatterlist info of the buffer to unmap

enum dma_data_direction direction [in] direction of DMA transfer

Description

This unmaps a DMA mapping for **attached** obtained by `dma_buf_map_attachment()`.

void **dma_buf_move_notify**(struct dma_buf * dmabuf)
notify attachments that DMA-buf is moving

Parameters

struct dma_buf * dmabuf [in] buffer which is moving

Description

Informs all attachmenst that they need to destroy and recreated all their mappings.

int **dma_buf_begin_cpu_access**(struct dma_buf * dmabuf, enum dma_data_direction direction)
Must be called before accessing a dma_buf from the cpu in the kernel context. Calls begin_cpu_access to allow exporter-specific preparations. Coherency is only guaranteed in the specified range for the specified access direction.

Parameters

struct dma_buf * dmabuf [in] buffer to prepare cpu access for.

enum dma_data_direction direction [in] length of range for cpu access.

Description

After the cpu access is complete the caller should call `dma_buf_end_cpu_access()`. Only when cpu access is braketed by both calls is it guaranteed to be coherent with other DMA access.

Can return negative error values, returns 0 on success.

int **dma_buf_end_cpu_access**(struct dma_buf * dmabuf, enum dma_data_direction direction)
Must be called after accessing a dma_buf from the cpu in the kernel context. Calls end_cpu_access to allow exporter-specific actions. Coherency is only guaranteed in the specified range for the specified access direction.

Parameters

struct dma_buf * dmabuf [in] buffer to complete cpu access for.

enum dma_data_direction direction [in] length of range for cpu access.

Description

This terminates CPU access started with `dma_buf_begin_cpu_access()`.

Can return negative error values, returns 0 on success.

int **dma_buf_mmap**(struct dma_buf * dmabuf, struct vm_area_struct * vma, unsigned long pgoff)
Setup up a userspace mmap with the given vma

Parameters

struct dma_buf * dmabuf [in] buffer that should back the vma

struct vm_area_struct * vma [in] vma for the mmap

unsigned long pgoff [in] offset in pages where this mmap should start within the dma-buf buffer.

Description

This function adjusts the passed in vma so that it points at the file of the dma_buf operation. It also adjusts the starting pgoff and does bounds checking on the size of the vma. Then it calls the exporters mmap function to set up the mapping.

Can return negative error values, returns 0 on success.

void * **dma_buf_vmap**(struct dma_buf * dmabuf)
Create virtual mapping for the buffer object into kernel address space. Same restrictions as for vmap and friends apply.

Parameters

struct dma_buf * dmabuf [in] buffer to vmap

Description

This call may fail due to lack of virtual mapping address space. These calls are optional in drivers. The intended use for them is for mapping objects linear in kernel space for high use objects. Please attempt to use kmap/kunmap before thinking about these interfaces.

Returns NULL on error.

void **dma_buf_vunmap**(struct dma_buf * dmabuf, void * vaddr)
Unmap a vmap obtained by dma_buf_vmap.

Parameters

struct dma_buf * dmabuf [in] buffer to vunmap

void * vaddr [in] vmap to vunmap

struct **dma_buf_ops**
operations possible on struct dma_buf

Definition

```
struct dma_buf_ops {
  bool cache_sgt_mapping;
  int (*attach)(struct dma_buf *, struct dma_buf_attachment *);
  void (*detach)(struct dma_buf *, struct dma_buf_attachment *);
  int (*pin)(struct dma_buf_attachment *attach);
  void (*unpin)(struct dma_buf_attachment *attach);
  struct sg_table * (*map_dma_buf)(struct dma_buf_attachment *, enum dma_
↪data_direction);
  void (*unmap_dma_buf)(struct dma_buf_attachment *,struct sg_table *,␣
↪enum dma_data_direction);
```

(continues on next page)

(continued from previous page)

```
  void (*release)(struct dma_buf *);
  int (*begin_cpu_access)(struct dma_buf *, enum dma_data_direction);
  int (*end_cpu_access)(struct dma_buf *, enum dma_data_direction);
  int (*mmap)(struct dma_buf *, struct vm_area_struct *vma);
  void *(*vmap)(struct dma_buf *);
  void (*vunmap)(struct dma_buf *, void *vaddr);
};
```

Members

cache_sgt_mapping If true the framework will cache the first mapping made for each attachment. This avoids creating mappings for attachments multiple times.

attach This is called from `dma_buf_attach()` to make sure that a given `dma_buf_attachment.dev` can access the provided `dma_buf`. Exporters which support buffer objects in special locations like VRAM or device-specific carve-out areas should check whether the buffer could be move to system memory (or directly accessed by the provided device), and otherwise need to fail the attach operation.

The exporter should also in general check whether the current allocation fulfills the DMA constraints of the new device. If this is not the case, and the allocation cannot be moved, it should also fail the attach operation.

Any exporter-private housekeeping data can be stored in the `dma_buf_attachment.priv` pointer.

This callback is optional.

Returns:

0 on success, negative error code on failure. It might return -EBUSY to signal that backing storage is already allocated and incompatible with the requirements of requesting device.

detach This is called by `dma_buf_detach()` to release a `dma_buf_attachment`. Provided so that exporters can clean up any housekeeping for an `dma_buf_attachment`.

This callback is optional.

pin This is called by dma_buf_pin and lets the exporter know that the DMA-buf can't be moved any more.

This is called with the dmabuf->resv object locked and is mutual exclusive with **cache_sgt_mapping**.

This callback is optional and should only be used in limited use cases like scanout and not for temporary pin operations.

Returns:

0 on success, negative error code on failure.

unpin This is called by dma_buf_unpin and lets the exporter know that the DMA-buf can be moved again.

This is called with the dmabuf->resv object locked and is mutual exclusive with **cache_sgt_mapping**.

This callback is optional.

map_dma_buf This is called by `dma_buf_map_attachment()` and is used to map a shared `dma_buf` into device address space, and it is mandatory. It can only be called if **attach** has been called successfully.

This call may sleep, e.g. when the backing storage first needs to be allocated, or moved to a location suitable for all currently attached devices.

Note that any specific buffer attributes required for this function should get added to device_dma_parameters accessible via `device.dma_params` from the `dma_buf_attachment`. The **attach** callback should also check these constraints.

If this is being called for the first time, the exporter can now choose to scan through the list of attachments for this buffer, collate the requirements of the attached devices, and choose an appropriate backing storage for the buffer.

Based on enum dma_data_direction, it might be possible to have multiple users accessing at the same time (for reading, maybe), or any other kind of sharing that the exporter might wish to make available to buffer-users.

This is always called with the dmabuf->resv object locked when the dynamic_mapping flag is true.

Returns:

A `sg_table` scatter list of or the backing storage of the DMA buffer, already mapped into the device address space of the `device` attached with the provided `dma_buf_attachment`.

On failure, returns a negative error value wrapped into a pointer. May also return -EINTR when a signal was received while being blocked.

unmap_dma_buf This is called by `dma_buf_unmap_attachment()` and should unmap and release the `sg_table` allocated in **map_dma_buf**, and it is mandatory. For static dma_buf handling this might also unpins the backing storage if this is the last mapping of the DMA buffer.

release Called after the last dma_buf_put to release the `dma_buf`, and mandatory.

begin_cpu_access This is called from `dma_buf_begin_cpu_access()` and allows the exporter to ensure that the memory is actually available for cpu access - the exporter might need to allocate or swap-in and pin the backing storage. The exporter also needs to ensure that cpu access is coherent for the access direction. The direction can be used by the exporter to optimize the cache flushing, i.e. access with a different direction (read instead of write) might return stale or even bogus data (e.g. when the exporter needs to copy the data to temporary storage).

This callback is optional.

FIXME: This is both called through the DMA_BUF_IOCTL_SYNC command from userspace (where storage shouldn't be pinned to avoid handing de-factor mlock rights to userspace) and for the kernel-internal users of the various kmap interfaces, where the backing storage must be pinned to guarantee

that the atomic kmap calls can succeed. Since there's no in-kernel users of the kmap interfaces yet this isn't a real problem.

Returns:

0 on success or a negative error code on failure. This can for example fail when the backing storage can't be allocated. Can also return -ERESTARTSYS or -EINTR when the call has been interrupted and needs to be restarted.

end_cpu_access This is called from `dma_buf_end_cpu_access()` when the importer is done accessing the CPU. The exporter can use this to flush caches and unpin any resources pinned in **begin_cpu_access**. The result of any dma_buf kmap calls after end_cpu_access is undefined.

This callback is optional.

Returns:

0 on success or a negative error code on failure. Can return -ERESTARTSYS or -EINTR when the call has been interrupted and needs to be restarted.

mmap This callback is used by the `dma_buf_mmap()` function

Note that the mapping needs to be incoherent, userspace is expected to braket CPU access using the DMA_BUF_IOCTL_SYNC interface.

Because dma-buf buffers have invariant size over their lifetime, the dma-buf core checks whether a vma is too large and rejects such mappings. The exporter hence does not need to duplicate this check. Drivers do not need to check this themselves.

If an exporter needs to manually flush caches and hence needs to fake coherency for mmap support, it needs to be able to zap all the ptes pointing at the backing storage. Now linux mm needs a struct address_space associated with the struct file stored in vma->vm_file to do that with the function unmap_mapping_range. But the dma_buf framework only backs every dma_buf fd with the anon_file struct file, i.e. all dma_bufs share the same file.

Hence exporters need to setup their own file (and address_space) association by setting vma->vm_file and adjusting vma->vm_pgoff in the dma_buf mmap callback. In the specific case of a gem driver the exporter could use the shmem file already provided by gem (and set vm_pgoff = 0). Exporters can then zap ptes by unmapping the corresponding range of the struct address_space associated with their own file.

This callback is optional.

Returns:

0 on success or a negative error code on failure.

vmap [optional] creates a virtual mapping for the buffer into kernel address space. Same restrictions as for vmap and friends apply.

vunmap [optional] unmaps a vmap from the buffer

struct **dma_buf**
shared buffer object

Definition

```
struct dma_buf {
  size_t size;
  struct file *file;
  struct list_head attachments;
  const struct dma_buf_ops *ops;
  struct mutex lock;
  unsigned vmapping_counter;
  void *vmap_ptr;
  const char *exp_name;
  const char *name;
  struct module *owner;
  struct list_head list_node;
  void *priv;
  struct dma_resv *resv;
  wait_queue_head_t poll;
  struct dma_buf_poll_cb_t {
    struct dma_fence_cb cb;
    wait_queue_head_t *poll;
    __poll_t active;
  } cb_excl, cb_shared;
};
```

Members

size size of the buffer

file file pointer used for sharing buffers across, and for refcounting.

attachments list of dma_buf_attachment that denotes all devices attached, protected by dma_resv lock.

ops dma_buf_ops associated with this buffer object.

lock used internally to serialize list manipulation, attach/detach and vmap/unmap

vmapping_counter used internally to refcnt the vmaps

vmap_ptr the current vmap ptr if vmapping_counter > 0

exp_name name of the exporter; useful for debugging.

name userspace-provided name; useful for accounting and debugging, protected by **resv**.

owner pointer to exporter module; used for refcounting when exporter is a kernel module.

list_node node for dma_buf accounting and debugging.

priv exporter specific private data for this buffer object.

resv reservation object linked to this dma-buf

poll for userspace poll support

cb_excl for userspace poll support

cb_shared for userspace poll support

Description

This represents a shared buffer, created by calling `dma_buf_export()`. The userspace representation is a normal file descriptor, which can be created by calling `dma_buf_fd()`.

Shared dma buffers are reference counted using `dma_buf_put()` and `get_dma_buf()`.

Device DMA access is handled by the separate `struct dma_buf_attachment`.

struct **dma_buf_attach_ops**
importer operations for an attachment

Definition

```
struct dma_buf_attach_ops {
  bool allow_peer2peer;
  void (*move_notify)(struct dma_buf_attachment *attach);
};
```

Members

allow_peer2peer If this is set to true the importer must be able to handle peer resources without struct pages.

move_notify [optional] notification that the DMA-buf is moving

If this callback is provided the framework can avoid pinning the backing store while mappings exists.

This callback is called with the lock of the reservation object associated with the dma_buf held and the mapping function must be called with this lock held as well. This makes sure that no mapping is created concurrently with an ongoing move operation.

Mappings stay valid and are not directly affected by this callback. But the DMA-buf can now be in a different physical location, so all mappings should be destroyed and re-created as soon as possible.

New mappings can be created after this callback returns, and will point to the new location of the DMA-buf.

Description

Attachment operations implemented by the importer.

struct **dma_buf_attachment**
holds device-buffer attachment data

Definition

```
struct dma_buf_attachment {
  struct dma_buf *dmabuf;
  struct device *dev;
  struct list_head node;
  struct sg_table *sgt;
  enum dma_data_direction dir;
  bool peer2peer;
  const struct dma_buf_attach_ops *importer_ops;
  void *importer_priv;
```

(continues on next page)

(continued from previous page)

```
  void *priv;
};
```

Members

dmabuf buffer for this attachment.

dev device attached to the buffer.

node list of dma_buf_attachment, protected by dma_resv lock of the dmabuf.

sgt cached mapping.

dir direction of cached mapping.

peer2peer true if the importer can handle peer resources without pages.

importer_ops importer operations for this attachment, if provided dma_buf_map/unmap_attachment() must be called with the dma_resv lock held.

importer_priv importer specific attachment data.

priv exporter specific attachment data.

Description

This structure holds the attachment information between the dma_buf buffer and its user device(s). The list contains one attachment struct per device attached to the buffer.

An attachment is created by calling `dma_buf_attach()`, and released again by calling `dma_buf_detach()`. The DMA mapping itself needed to initiate a transfer is created by `dma_buf_map_attachment()` and freed again by calling `dma_buf_unmap_attachment()`.

struct **dma_buf_export_info**
holds information needed to export a dma_buf

Definition

```
struct dma_buf_export_info {
  const char *exp_name;
  struct module *owner;
  const struct dma_buf_ops *ops;
  size_t size;
  int flags;
  struct dma_resv *resv;
  void *priv;
};
```

Members

exp_name name of the exporter - useful for debugging.

owner pointer to exporter module - used for refcounting kernel module

ops Attach allocator-defined dma buf ops to the new buffer

size Size of the buffer

flags mode flags for the file

resv reservation-object, NULL to allocate default one

priv Attach private data of allocator to this buffer

Description

This structure holds the information required to export the buffer. Used with `dma_buf_export()` only.

DEFINE_DMA_BUF_EXPORT_INFO(name)
helper macro for exporters

Parameters

name export-info name

Description

DEFINE_DMA_BUF_EXPORT_INFO macro defines the `struct dma_buf_export_info`, zeroes it out and pre-populates exp_name in it.

void **get_dma_buf**(struct dma_buf * dmabuf)
convenience wrapper for get_file.

Parameters

struct dma_buf * dmabuf [in] pointer to dma_buf

Description

Increments the reference count on the dma-buf, needed in case of drivers that either need to create additional references to the dmabuf on the kernel side. For example, an exporter that needs to keep a dmabuf ptr so that subsequent exports don't create a new dmabuf.

bool **dma_buf_is_dynamic**(struct dma_buf * dmabuf)
check if a DMA-buf uses dynamic mappings.

Parameters

struct dma_buf * dmabuf the DMA-buf to check

Description

Returns true if a DMA-buf exporter wants to be called with the dma_resv locked for the map/unmap callbacks, false if it doesn't wants to be called with the lock held.

bool **dma_buf_attachment_is_dynamic**(struct dma_buf_attachment * attach)
check if a DMA-buf attachment uses dynamic mappinsg

Parameters

struct dma_buf_attachment * attach the DMA-buf attachment to check

Description

Returns true if a DMA-buf importer wants to call the map/unmap functions with the dma_resv lock held.

10.2 Reservation Objects

The reservation object provides a mechanism to manage shared and exclusive fences associated with a buffer. A reservation object can have attached one exclusive fence (normally associated with write operations) or N shared fences (read operations). The RCU mechanism is used to protect read access to fences from locked write-side updates.

void **dma_resv_init**(struct dma_resv * obj)
initialize a reservation object

Parameters

struct dma_resv * obj the reservation object

void **dma_resv_fini**(struct dma_resv * obj)
destroys a reservation object

Parameters

struct dma_resv * obj the reservation object

int **dma_resv_reserve_shared**(struct dma_resv * obj, unsigned int num_fences)
Reserve space to add shared fences to a dma_resv.

Parameters

struct dma_resv * obj reservation object

unsigned int num_fences number of fences we want to add

Description

Should be called before `dma_resv_add_shared_fence()`. Must be called with obj->lock held.

RETURNS Zero for success, or -errno

void **dma_resv_add_shared_fence**(struct dma_resv * obj, struct dma_fence * fence)
Add a fence to a shared slot

Parameters

struct dma_resv * obj the reservation object

struct dma_fence * fence the shared fence to add

Description

Add a fence to a shared slot, obj->lock must be held, and `dma_resv_reserve_shared()` has been called.

void **dma_resv_add_excl_fence**(struct dma_resv * obj, struct dma_fence * fence)
Add an exclusive fence.

Parameters

struct dma_resv * obj the reservation object

struct dma_fence * fence the shared fence to add

Description

Add a fence to the exclusive slot. The obj->lock must be held.

int **dma_resv_copy_fences**(struct dma_resv * dst, struct dma_resv * src)
Copy all fences from src to dst.

Parameters

struct dma_resv * dst the destination reservation object

struct dma_resv * src the source reservation object

Description

Copy all fences from src to dst. dst-lock must be held.

int **dma_resv_get_fences_rcu**(struct dma_resv * obj, struct dma_fence ** pfence_excl, unsigned * pshared_count, struct dma_fence *** pshared)
Get an object's shared and exclusive fences without update side lock held

Parameters

struct dma_resv * obj the reservation object

struct dma_fence ** pfence_excl the returned exclusive fence (or NULL)

unsigned * pshared_count the number of shared fences returned

struct dma_fence * pshared** the array of shared fence ptrs returned (array is krealloc'd to the required size, and must be freed by caller)

Description

Retrieve all fences from the reservation object. If the pointer for the exclusive fence is not specified the fence is put into the array of the shared fences as well. Returns either zero or -ENOMEM.

long **dma_resv_wait_timeout_rcu**(struct dma_resv * obj, bool wait_all, bool intr, unsigned long timeout)
Wait on reservation's objects shared and/or exclusive fences.

Parameters

struct dma_resv * obj the reservation object

bool wait_all if true, wait on all fences, else wait on just exclusive fence

bool intr if true, do interruptible wait

unsigned long timeout timeout value in jiffies or zero to return immediately

Description

RETURNS Returns -ERESTARTSYS if interrupted, 0 if the wait timed out, or greater than zer on success.

bool **dma_resv_test_signaled_rcu**(struct dma_resv * obj, bool test_all)
Test if a reservation object's fences have been signaled.

Parameters

struct dma_resv * obj the reservation object

bool test_all if true, test all fences, otherwise only test the exclusive fence

Description

RETURNS true if all fences signaled, else false

struct **dma_resv_list**
a list of shared fences

Definition

```
struct dma_resv_list {
  struct rcu_head rcu;
  u32 shared_count, shared_max;
  struct dma_fence __rcu *shared[];
};
```

Members

rcu for internal use

shared_count table of shared fences

shared_max for growing shared fence table

shared shared fence table

struct **dma_resv**
a reservation object manages fences for a buffer

Definition

```
struct dma_resv {
  struct ww_mutex lock;
  seqcount_t seq;
  struct dma_fence __rcu *fence_excl;
  struct dma_resv_list __rcu *fence;
};
```

Members

lock update side lock

seq sequence count for managing RCU read-side synchronization

fence_excl the exclusive fence, if there is one currently

fence list of current shared fences

struct dma_resv_list * **dma_resv_get_list**(struct dma_resv * obj)
get the reservation object's shared fence list, with update-side lock held

Parameters

struct dma_resv * obj the reservation object

Description

Returns the shared fence list. Does NOT take references to the fence. The obj->lock must be held.

int **dma_resv_lock**(struct dma_resv * obj, struct ww_acquire_ctx * ctx)
lock the reservation object

Parameters

struct dma_resv * obj the reservation object

struct ww_acquire_ctx * ctx the locking context

Description

Locks the reservation object for exclusive access and modification. Note, that the lock is only against other writers, readers will run concurrently with a writer under RCU. The seqlock is used to notify readers if they overlap with a writer.

As the reservation object may be locked by multiple parties in an undefined order, a #ww_acquire_ctx is passed to unwind if a cycle is detected. See ww_mutex_lock() and ww_acquire_init(). A reservation object may be locked by itself by passing NULL as **ctx**.

int **dma_resv_lock_interruptible**(struct dma_resv * obj, struct ww_acquire_ctx * ctx)
lock the reservation object

Parameters

struct dma_resv * obj the reservation object

struct ww_acquire_ctx * ctx the locking context

Description

Locks the reservation object interruptible for exclusive access and modification. Note, that the lock is only against other writers, readers will run concurrently with a writer under RCU. The seqlock is used to notify readers if they overlap with a writer.

As the reservation object may be locked by multiple parties in an undefined order, a #ww_acquire_ctx is passed to unwind if a cycle is detected. See ww_mutex_lock() and ww_acquire_init(). A reservation object may be locked by itself by passing NULL as **ctx**.

void **dma_resv_lock_slow**(struct dma_resv * obj, struct ww_acquire_ctx * ctx)
slowpath lock the reservation object

Parameters

struct dma_resv * obj the reservation object

struct ww_acquire_ctx * ctx the locking context

Description

Acquires the reservation object after a die case. This function will sleep until the lock becomes available. See `dma_resv_lock()` as well.

int **dma_resv_lock_slow_interruptible**(struct dma_resv * obj, struct ww_acquire_ctx * ctx)
slowpath lock the reservation object, interruptible

Parameters

struct dma_resv * obj the reservation object

struct ww_acquire_ctx * ctx the locking context

Description

Acquires the reservation object interruptible after a die case. This function will sleep until the lock becomes available. See `dma_resv_lock_interruptible()` as well.

bool **dma_resv_trylock**(struct dma_resv * obj)
trylock the reservation object

Parameters

struct dma_resv * obj the reservation object

Description

Tries to lock the reservation object for exclusive access and modification. Note, that the lock is only against other writers, readers will run concurrently with a writer under RCU. The seqlock is used to notify readers if they overlap with a writer.

Also note that since no context is provided, no deadlock protection is possible.

Returns true if the lock was acquired, false otherwise.

bool **dma_resv_is_locked**(struct dma_resv * obj)
is the reservation object locked

Parameters

struct dma_resv * obj the reservation object

Description

Returns true if the mutex is locked, false if unlocked.

struct ww_acquire_ctx * **dma_resv_locking_ctx**(struct dma_resv * obj)
returns the context used to lock the object

Parameters

struct dma_resv * obj the reservation object

Description

Returns the context used to lock a reservation object or NULL if no context was used or the object is not locked at all.

void **dma_resv_unlock**(struct dma_resv * obj)
unlock the reservation object

Parameters

struct dma_resv * obj the reservation object

Description

Unlocks the reservation object following exclusive access.

struct dma_fence * **dma_resv_get_excl**(struct dma_resv * obj)
get the reservation object's exclusive fence, with update-side lock held

Parameters

struct dma_resv * obj the reservation object

Description

Returns the exclusive fence (if any). Does NOT take a reference. Writers must hold obj->lock, readers may only hold a RCU read side lock.

RETURNS The exclusive fence or NULL

struct dma_fence * **dma_resv_get_excl_rcu**(struct dma_resv * obj)
get the reservation object's exclusive fence, without lock held.

Parameters

struct dma_resv * obj the reservation object

Description

If there is an exclusive fence, this atomically increments it's reference count and returns it.

RETURNS The exclusive fence or NULL if none

10.3 DMA Fences

DMA fences, represented by `struct dma_fence`, are the kernel internal synchronization primitive for DMA operations like GPU rendering, video encoding/decoding, or displaying buffers on a screen.

A fence is initialized using `dma_fence_init()` and completed using `dma_fence_signal()`. Fences are associated with a context, allocated through `dma_fence_context_alloc()`, and all fences on the same context are fully ordered.

Since the purposes of fences is to facilitate cross-device and cross-application synchronization, there's multiple ways to use one:

- Individual fences can be exposed as a `sync_file`, accessed as a file descriptor from userspace, created by calling `sync_file_create()`. This is called explicit fencing, since userspace passes around explicit synchronization points.
- Some subsystems also have their own explicit fencing primitives, like `drm_syncobj`. Compared to `sync_file`, a `drm_syncobj` allows the underlying fence to be updated.
- Then there's also implicit fencing, where the synchronization points are implicitly passed around as part of shared `dma_buf` instances. Such implicit fences are stored in `struct dma_resv` through the `dma_buf.resv` pointer.

10.3.1 DMA Fences Functions Reference

struct dma_fence * **dma_fence_get_stub**(void)
return a signaled fence

Parameters

void no arguments

Description

Return a stub fence which is already signaled.

u64 **dma_fence_context_alloc**(unsigned num)
allocate an array of fence contexts

Parameters

unsigned num amount of contexts to allocate

Description

This function will return the first index of the number of fence contexts allocated. The fence context is used for setting `dma_fence.context` to a unique number by passing the context to `dma_fence_init()`.

int **dma_fence_signal_locked**(struct dma_fence * fence)
signal completion of a fence

Parameters

struct dma_fence * fence the fence to signal

Description

Signal completion for software callbacks on a fence, this will unblock `dma_fence_wait()` calls and run all the callbacks added with `dma_fence_add_callback()`. Can be called multiple times, but since a fence can only go from the unsignaled to the signaled state and not back, it will only be effective the first time.

Unlike `dma_fence_signal()`, this function must be called with `dma_fence.lock` held.

Returns 0 on success and a negative error value when **fence** has been signalled already.

int **dma_fence_signal**(struct dma_fence * fence)
signal completion of a fence

Parameters

struct dma_fence * fence the fence to signal

Description

Signal completion for software callbacks on a fence, this will unblock `dma_fence_wait()` calls and run all the callbacks added with `dma_fence_add_callback()`. Can be called multiple times, but since a fence can only go from the unsignaled to the signaled state and not back, it will only be effective the first time.

Returns 0 on success and a negative error value when **fence** has been signalled already.

signed long **dma_fence_wait_timeout**(struct dma_fence * fence, bool intr, signed long timeout)
sleep until the fence gets signaled or until timeout elapses

Parameters

struct dma_fence * fence the fence to wait on

bool intr if true, do an interruptible wait

signed long timeout timeout value in jiffies, or MAX_SCHEDULE_TIMEOUT

Description

Returns -ERESTARTSYS if interrupted, 0 if the wait timed out, or the remaining timeout in jiffies on success. Other error values may be returned on custom implementations.

Performs a synchronous wait on this fence. It is assumed the caller directly or indirectly (buf-mgr between reservation and committing) holds a reference to the fence, otherwise the fence might be freed before return, resulting in undefined behavior.

See also `dma_fence_wait()` and `dma_fence_wait_any_timeout()`.

void **dma_fence_release**(struct kref * kref)
default relese function for fences

Parameters

struct kref * kref `dma_fence.recfount`

Description

This is the default release functions for `dma_fence`. Drivers shouldn't call this directly, but instead call `dma_fence_put()`.

void **dma_fence_free**(struct dma_fence * fence)
default release function for `dma_fence`.

Parameters

struct dma_fence * fence fence to release

Description

This is the default implementation for `dma_fence_ops.release`. It calls kfree_rcu() on **fence**.

void **dma_fence_enable_sw_signaling**(struct dma_fence * fence)
enable signaling on fence

Parameters

struct dma_fence * fence the fence to enable

Description

This will request for sw signaling to be enabled, to make the fence complete as soon as possible. This calls `dma_fence_ops.enable_signaling` internally.

int **dma_fence_add_callback**(struct dma_fence * fence, struct dma_fence_cb * cb, dma_fence_func_t func)
add a callback to be called when the fence is signaled

Parameters

struct dma_fence * fence the fence to wait on

struct dma_fence_cb * cb the callback to register

dma_fence_func_t func the function to call

Description

cb will be initialized by `dma_fence_add_callback()`, no initialization by the caller is required. Any number of callbacks can be registered to a fence, but a callback can only be registered to one fence at a time.

Note that the callback can be called from an atomic context. If fence is already signaled, this function will return -ENOENT (and not call the callback).

Add a software callback to the fence. Same restrictions apply to refcount as it does to `dma_fence_wait()`, however the caller doesn't need to keep a refcount to fence afterward `dma_fence_add_callback()` has returned: when software access is enabled, the creator of the fence is required to keep the fence alive until after it signals with `dma_fence_signal()`. The callback itself can be called from irq context.

Returns 0 in case of success, -ENOENT if the fence is already signaled and -EINVAL in case of error.

int **dma_fence_get_status**(struct dma_fence * fence)
returns the status upon completion

Parameters

struct dma_fence * fence the dma_fence to query

Description

This wraps `dma_fence_get_status_locked()` to return the error status condition on a signaled fence. See `dma_fence_get_status_locked()` for more details.

Returns 0 if the fence has not yet been signaled, 1 if the fence has been signaled without an error condition, or a negative error code if the fence has been completed in err.

bool **dma_fence_remove_callback**(struct dma_fence * fence, struct dma_fence_cb * cb)
remove a callback from the signaling list

Parameters

struct dma_fence * fence the fence to wait on

struct dma_fence_cb * cb the callback to remove

Description

Remove a previously queued callback from the fence. This function returns true if the callback is successfully removed, or false if the fence has already been signaled.

WARNING: Cancelling a callback should only be done if you really know what you're doing, since deadlocks and race conditions could occur all too easily. For this reason, it should only ever be done on hardware lockup recovery, with a reference held to the fence.

Behaviour is undefined if **cb** has not been added to **fence** using `dma_fence_add_callback()` beforehand.

signed long **dma_fence_default_wait**(struct dma_fence * fence, bool intr, signed long timeout)
default sleep until the fence gets signaled or until timeout elapses

Parameters

struct dma_fence * fence the fence to wait on

bool intr if true, do an interruptible wait

signed long timeout timeout value in jiffies, or MAX_SCHEDULE_TIMEOUT

Description

Returns -ERESTARTSYS if interrupted, 0 if the wait timed out, or the remaining timeout in jiffies on success. If timeout is zero the value one is returned if the fence is already signaled for consistency with other functions taking a jiffies timeout.

signed long **dma_fence_wait_any_timeout**(struct dma_fence ** fences, uint32_t count, bool intr, signed long timeout, uint32_t * idx)
sleep until any fence gets signaled or until timeout elapses

Parameters

struct dma_fence ** fences array of fences to wait on

uint32_t count number of fences to wait on

bool intr if true, do an interruptible wait

signed long timeout timeout value in jiffies, or MAX_SCHEDULE_TIMEOUT

uint32_t * idx used to store the first signaled fence index, meaningful only on positive return

Description

Returns -EINVAL on custom fence wait implementation, -ERESTARTSYS if interrupted, 0 if the wait timed out, or the remaining timeout in jiffies on success.

Synchronous waits for the first fence in the array to be signaled. The caller needs to hold a reference to all fences in the array, otherwise a fence might be freed before return, resulting in undefined behavior.

See also `dma_fence_wait()` and `dma_fence_wait_timeout()`.

void **dma_fence_init**(struct dma_fence * fence, const struct dma_fence_ops * ops, spinlock_t * lock, u64 context, u64 seqno)
Initialize a custom fence.

Parameters

struct dma_fence * fence the fence to initialize

const struct dma_fence_ops * ops the dma_fence_ops for operations on this fence

spinlock_t * lock the irqsafe spinlock to use for locking this fence

u64 context the execution context this fence is run on

u64 seqno a linear increasing sequence number for this context

Description

Initializes an allocated fence, the caller doesn't have to keep its refcount after committing with this fence, but it will need to hold a refcount again if `dma_fence_ops.enable_signaling` gets called.

context and seqno are used for easy comparison between fences, allowing to check which fence is later by simply using `dma_fence_later()`.

struct **dma_fence**
software synchronization primitive

Definition

```
struct dma_fence {
  spinlock_t *lock;
  const struct dma_fence_ops *ops;
  union {
    struct list_head cb_list;
    ktime_t timestamp;
    struct rcu_head rcu;
  };
  u64 context;
  u64 seqno;
  unsigned long flags;
  struct kref refcount;
  int error;
};
```

Members

lock spin_lock_irqsave used for locking

ops dma_fence_ops associated with this fence

{unnamed_union} anonymous

cb_list list of all callbacks to call

timestamp Timestamp when the fence was signaled.

rcu used for releasing fence with kfree_rcu

context execution context this fence belongs to, returned by `dma_fence_context_alloc()`

seqno the sequence number of this fence inside the execution context, can be compared to decide which fence would be signaled later.

flags A mask of DMA_FENCE_FLAG_* defined below

refcount refcount for this fence

error Optional, only valid if < 0, must be set before calling dma_fence_signal, indicates that the fence has completed with an error.

Description

the flags member must be manipulated and read using the appropriate atomic ops (bit_*), so taking the spinlock will not be needed most of the time.

DMA_FENCE_FLAG_SIGNALED_BIT - fence is already signaled DMA_FENCE_FLAG_TIMESTAMP_BIT - timestamp recorded for fence signaling DMA_FENCE_FLAG_ENABLE_SIGNAL_BIT - enable_signaling might have been called DMA_FENCE_FLAG_USER_BITS - start of the unused bits, can be used by the implementer of the fence for its own purposes. Can be used in different ways by different fence implementers, so do not rely on this.

Since atomic bitops are used, this is not guaranteed to be the case. Particularly, if the bit was set, but dma_fence_signal was called right before this bit was set, it would have been able to set the DMA_FENCE_FLAG_SIGNALED_BIT, before enable_signaling was called. Adding a check for DMA_FENCE_FLAG_SIGNALED_BIT after setting DMA_FENCE_FLAG_ENABLE_SIGNAL_BIT closes this race, and makes sure that after dma_fence_signal was called, any enable_signaling call will have either been completed, or never called at all.

struct **dma_fence_cb**
callback for `dma_fence_add_callback()`

Definition

```
struct dma_fence_cb {
  struct list_head node;
  dma_fence_func_t func;
};
```

Members

node used by `dma_fence_add_callback()` to append this struct to fence::cb_list

func dma_fence_func_t to call

Description

This struct will be initialized by `dma_fence_add_callback()`, additional data can be passed along by embedding dma_fence_cb in another struct.

struct **dma_fence_ops**
operations implemented for fence

Definition

```
struct dma_fence_ops {
  bool use_64bit_seqno;
  const char * (*get_driver_name)(struct dma_fence *fence);
  const char * (*get_timeline_name)(struct dma_fence *fence);
  bool (*enable_signaling)(struct dma_fence *fence);
  bool (*signaled)(struct dma_fence *fence);
  signed long (*wait)(struct dma_fence *fence, bool intr, signed long␣
↪timeout);
```

(continues on next page)

(continued from previous page)

```
  void (*release)(struct dma_fence *fence);
  void (*fence_value_str)(struct dma_fence *fence, char *str, int size);
  void (*timeline_value_str)(struct dma_fence *fence, char *str, int size);
};
```

Members

use_64bit_seqno True if this dma_fence implementation uses 64bit seqno, false otherwise.

get_driver_name Returns the driver name. This is a callback to allow drivers to compute the name at runtime, without having it to store permanently for each fence, or build a cache of some sort.

This callback is mandatory.

get_timeline_name Return the name of the context this fence belongs to. This is a callback to allow drivers to compute the name at runtime, without having it to store permanently for each fence, or build a cache of some sort.

This callback is mandatory.

enable_signaling Enable software signaling of fence.

For fence implementations that have the capability for hw->hw signaling, they can implement this op to enable the necessary interrupts, or insert commands into cmdstream, etc, to avoid these costly operations for the common case where only hw->hw synchronization is required. This is called in the first `dma_fence_wait()` or `dma_fence_add_callback()` path to let the fence implementation know that there is another driver waiting on the signal (ie. hw->sw case).

This function can be called from atomic context, but not from irq context, so normal spinlocks can be used.

A return value of false indicates the fence already passed, or some failure occurred that made it impossible to enable signaling. True indicates successful enabling.

`dma_fence.error` may be set in enable_signaling, but only when false is returned.

Since many implementations can call `dma_fence_signal()` even when before **enable_signaling** has been called there' s a race window, where the `dma_fence_signal()` might result in the final fence reference being released and its memory freed. To avoid this, implementations of this callback should grab their own reference using `dma_fence_get()`, to be released when the fence is signalled (through e.g. the interrupt handler).

This callback is optional. If this callback is not present, then the driver must always have signaling enabled.

signaled Peek whether the fence is signaled, as a fastpath optimization for e.g. `dma_fence_wait()` or `dma_fence_add_callback()`. Note that this callback does not need to make any guarantees beyond that a fence once indicates as signalled must always return true from this callback. This callback may return false even if the fence has completed already, in this

case information hasn’ t propogated throug the system yet. See also `dma_fence_is_signaled()`.

May set `dma_fence.error` if returning true.

This callback is optional.

wait Custom wait implementation, defaults to `dma_fence_default_wait()` if not set.

The dma_fence_default_wait implementation should work for any fence, as long as **enable_signaling** works correctly. This hook allows drivers to have an optimized version for the case where a process context is already available, e.g. if **enable_signaling** for the general case needs to set up a worker thread.

Must return -ERESTARTSYS if the wait is intr = true and the wait was interrupted, and remaining jiffies if fence has signaled, or 0 if wait timed out. Can also return other error values on custom implementations, which should be treated as if the fence is signaled. For example a hardware lockup could be reported like that.

This callback is optional.

release Called on destruction of fence to release additional resources. Can be called from irq context. This callback is optional. If it is NULL, then `dma_fence_free()` is instead called as the default implementation.

fence_value_str Callback to fill in free-form debug info specific to this fence, like the sequence number.

This callback is optional.

timeline_value_str Fills in the current value of the timeline as a string, like the sequence number. Note that the specific fence passed to this function should not matter, drivers should only use it to look up the corresponding timeline structures.

void **dma_fence_put**(struct dma_fence * fence)
decreases refcount of the fence

Parameters

struct dma_fence * fence fence to reduce refcount of

struct dma_fence * **dma_fence_get**(struct dma_fence * fence)
increases refcount of the fence

Parameters

struct dma_fence * fence fence to increase refcount of

Description

Returns the same fence, with refcount increased by 1.

struct dma_fence * **dma_fence_get_rcu**(struct dma_fence * fence)
get a fence from a dma_resv_list with rcu read lock

Parameters

struct dma_fence * fence fence to increase refcount of

Description

Function returns NULL if no refcount could be obtained, or the fence.

struct dma_fence * **dma_fence_get_rcu_safe**(struct dma_fence __rcu ** fencep)
acquire a reference to an RCU tracked fence

Parameters

struct dma_fence __rcu ** fencep pointer to fence to increase refcount of

Description

Function returns NULL if no refcount could be obtained, or the fence. This function handles acquiring a reference to a fence that may be reallocated within the RCU grace period (such as with SLAB_TYPESAFE_BY_RCU), so long as the caller is using RCU on the pointer to the fence.

An alternative mechanism is to employ a seqlock to protect a bunch of fences, such as used by struct dma_resv. When using a seqlock, the seqlock must be taken before and checked after a reference to the fence is acquired (as shown here).

The caller is required to hold the RCU read lock.

bool **dma_fence_is_signaled_locked**(struct dma_fence * fence)
Return an indication if the fence is signaled yet.

Parameters

struct dma_fence * fence the fence to check

Description

Returns true if the fence was already signaled, false if not. Since this function doesn't enable signaling, it is not guaranteed to ever return true if `dma_fence_add_callback()`, `dma_fence_wait()` or `dma_fence_enable_sw_signaling()` haven't been called before.

This function requires `dma_fence.lock` to be held.

See also `dma_fence_is_signaled()`.

bool **dma_fence_is_signaled**(struct dma_fence * fence)
Return an indication if the fence is signaled yet.

Parameters

struct dma_fence * fence the fence to check

Description

Returns true if the fence was already signaled, false if not. Since this function doesn't enable signaling, it is not guaranteed to ever return true if `dma_fence_add_callback()`, `dma_fence_wait()` or `dma_fence_enable_sw_signaling()` haven't been called before.

It's recommended for seqno fences to call dma_fence_signal when the operation is complete, it makes it possible to prevent issues from wraparound between time of issue and time of use by checking the return value of this function before calling hardware-specific wait instructions.

See also `dma_fence_is_signaled_locked()`.

bool **__dma_fence_is_later**(u64 f1, u64 f2, const struct dma_fence_ops * ops)
return if f1 is chronologically later than f2

Parameters

u64 f1 the first fence's seqno

u64 f2 the second fence's seqno from the same context

const struct dma_fence_ops * ops dma_fence_ops associated with the seqno

Description

Returns true if f1 is chronologically later than f2. Both fences must be from the same context, since a seqno is not common across contexts.

bool **dma_fence_is_later**(struct dma_fence * f1, struct dma_fence * f2)
return if f1 is chronologically later than f2

Parameters

struct dma_fence * f1 the first fence from the same context

struct dma_fence * f2 the second fence from the same context

Description

Returns true if f1 is chronologically later than f2. Both fences must be from the same context, since a seqno is not re-used across contexts.

struct dma_fence * **dma_fence_later**(struct dma_fence * f1, struct dma_fence * f2)
return the chronologically later fence

Parameters

struct dma_fence * f1 the first fence from the same context

struct dma_fence * f2 the second fence from the same context

Description

Returns NULL if both fences are signaled, otherwise the fence that would be signaled last. Both fences must be from the same context, since a seqno is not re-used across contexts.

int **dma_fence_get_status_locked**(struct dma_fence * fence)
returns the status upon completion

Parameters

struct dma_fence * fence the dma_fence to query

Description

Drivers can supply an optional error status condition before they signal the fence (to indicate whether the fence was completed due to an error rather than success). The value of the status condition is only valid if the fence has been signaled, `dma_fence_get_status_locked()` first checks the signal state before reporting the error status.

Returns 0 if the fence has not yet been signaled, 1 if the fence has been signaled without an error condition, or a negative error code if the fence has been completed in err.

void **dma_fence_set_error**(struct dma_fence * fence, int error)
flag an error condition on the fence

Parameters

struct dma_fence * fence the dma_fence

int error the error to store

Description

Drivers can supply an optional error status condition before they signal the fence, to indicate that the fence was completed due to an error rather than success. This must be set before signaling (so that the value is visible before any waiters on the signal callback are woken). This helper exists to help catching erroneous setting of #dma_fence.error.

signed long **dma_fence_wait**(struct dma_fence * fence, bool intr)
sleep until the fence gets signaled

Parameters

struct dma_fence * fence the fence to wait on

bool intr if true, do an interruptible wait

Description

This function will return -ERESTARTSYS if interrupted by a signal, or 0 if the fence was signaled. Other error values may be returned on custom implementations.

Performs a synchronous wait on this fence. It is assumed the caller directly or indirectly holds a reference to the fence, otherwise the fence might be freed before return, resulting in undefined behavior.

See also `dma_fence_wait_timeout()` and `dma_fence_wait_any_timeout()`.

10.3.2 Seqno Hardware Fences

struct seqno_fence * **to_seqno_fence**(struct dma_fence * fence)
cast a fence to a seqno_fence

Parameters

struct dma_fence * fence fence to cast to a seqno_fence

Description

Returns NULL if the fence is not a seqno_fence, or the seqno_fence otherwise.

void **seqno_fence_init**(struct seqno_fence * fence, spinlock_t * lock, struct dma_buf * sync_buf, uint32_t context, uint32_t seqno_ofs, uint32_t seqno, enum seqno_fence_condition cond, const struct dma_fence_ops * ops)
initialize a seqno fence

Parameters

struct seqno_fence * fence seqno_fence to initialize

spinlock_t * lock pointer to spinlock to use for fence

struct dma_buf * sync_buf buffer containing the memory location to signal on

uint32_t context the execution context this fence is a part of

uint32_t seqno_ofs the offset within **sync_buf**

uint32_t seqno the sequence # to signal on

enum seqno_fence_condition cond fence wait condition

const struct dma_fence_ops * ops the fence_ops for operations on this seqno fence

Description

This function initializes a struct seqno_fence with passed parameters, and takes a reference on sync_buf which is released on fence destruction.

A seqno_fence is a dma_fence which can complete in software when enable_signaling is called, but it also completes when (s32)((sync_buf)[seqno_ofs] - seqno) >= 0 is true

The seqno_fence will take a refcount on the sync_buf until it's destroyed, but actual lifetime of sync_buf may be longer if one of the callers take a reference to it.

Certain hardware have instructions to insert this type of wait condition in the command stream, so no intervention from software would be needed. This type of fence can be destroyed before completed, however a reference on the sync_buf dma-buf can be taken. It is encouraged to re-use the same dma-buf for sync_buf, since mapping or unmapping the sync_buf to the device's vm can be expensive.

It is recommended for creators of seqno_fence to call `dma_fence_signal()` before destruction. This will prevent possible issues from wraparound at time of issue vs time of check, since users can check `dma_fence_is_signaled()` before submitting instructions for the hardware to wait on the fence. However, when ops.enable_signaling is not called, it doesn't have to be done as soon as possible, just before there's any real danger of seqno wraparound.

10.3.3 DMA Fence Array

struct dma_fence_array * **dma_fence_array_create**(int num_fences, struct dma_fence ** fences, u64 context, unsigned seqno, bool signal_on_any)

Create a custom fence array

Parameters

int num_fences [in] number of fences to add in the array

struct dma_fence ** fences [in] array containing the fences

u64 context [in] fence context to use

unsigned seqno [in] sequence number to use

bool signal_on_any [in] signal on any fence in the array

Description

Allocate a dma_fence_array object and initialize the base fence with `dma_fence_init()`. In case of error it returns NULL.

The caller should allocate the fences array with num_fences size and fill it with the fences it wants to add to the object. Ownership of this array is taken and `dma_fence_put()` is used on each fence on release.

If **signal_on_any** is true the fence array signals if any fence in the array signals, otherwise it signals when all fences in the array signal.

bool **dma_fence_match_context**(struct dma_fence * fence, u64 context)
Check if all fences are from the given context

Parameters

struct dma_fence * fence [in] fence or fence array

u64 context [in] fence context to check all fences against

Description

Checks the provided fence or, for a fence array, all fences in the array against the given context. Returns false if any fence is from a different context.

struct **dma_fence_array_cb**
callback helper for fence array

Definition

```
struct dma_fence_array_cb {
  struct dma_fence_cb cb;
  struct dma_fence_array *array;
};
```

Members

cb fence callback structure for signaling

array reference to the parent fence array object

struct **dma_fence_array**
fence to represent an array of fences

Definition

```
struct dma_fence_array {
  struct dma_fence base;
  spinlock_t lock;
  unsigned num_fences;
  atomic_t num_pending;
  struct dma_fence **fences;
  struct irq_work work;
};
```

Members

base fence base class

lock spinlock for fence handling

num_fences number of fences in the array

num_pending fences in the array still pending

fences array of the fences

work internal irq_work function

bool **dma_fence_is_array**(struct dma_fence * fence)
check if a fence is from the array subsclass

Parameters

struct dma_fence * fence fence to test

Description

Return true if it is a dma_fence_array and false otherwise.

struct dma_fence_array * **to_dma_fence_array**(struct dma_fence * fence)
cast a fence to a dma_fence_array

Parameters

struct dma_fence * fence fence to cast to a dma_fence_array

Description

Returns NULL if the fence is not a dma_fence_array, or the dma_fence_array otherwise.

10.3.4 DMA Fence uABI/Sync File

struct sync_file * **sync_file_create**(struct dma_fence * fence)
creates a sync file

Parameters

struct dma_fence * fence fence to add to the sync_fence

Description

Creates a sync_file containg **fence**. This function acquires and additional reference of **fence** for the newly-created `sync_file`, if it succeeds. The sync_file can be released with fput(sync_file->file). Returns the sync_file or NULL in case of error.

struct dma_fence * **sync_file_get_fence**(int fd)
get the fence related to the sync_file fd

Parameters

int fd sync_file fd to get the fence from

Description

Ensures **fd** references a valid sync_file and returns a fence that represents all fence in the sync_file. On error NULL is returned.

struct **sync_file**
sync file to export to the userspace

Definition

```
struct sync_file {
  struct file             *file;
  char user_name[32];
#ifdef CONFIG_DEBUG_FS;
  struct list_head        sync_file_list;
#endif;
  wait_queue_head_t wq;
  unsigned long           flags;
  struct dma_fence        *fence;
  struct dma_fence_cb cb;
};
```

Members

file file representing this fence

user_name Name of the sync file provided by userspace, for merged fences. Otherwise generated through driver callbacks (in which case the entire array is 0).

sync_file_list membership in global file list

wq wait queue for fence signaling

flags flags for the sync_file

fence fence with the fences in the sync_file

cb fence callback information

Description

flags: POLL_ENABLED: whether userspace is currently poll()’ing or not

CHAPTER

ELEVEN

DEVICE LINKS

By default, the driver core only enforces dependencies between devices that are borne out of a parent/child relationship within the device hierarchy: When suspending, resuming or shutting down the system, devices are ordered based on this relationship, i.e. children are always suspended before their parent, and the parent is always resumed before its children.

Sometimes there is a need to represent device dependencies beyond the mere parent/child relationship, e.g. between siblings, and have the driver core automatically take care of them.

Secondly, the driver core by default does not enforce any driver presence dependencies, i.e. that one device must be bound to a driver before another one can probe or function correctly.

Often these two dependency types come together, so a device depends on another one both with regards to driver presence and with regards to suspend/resume and shutdown ordering.

Device links allow representation of such dependencies in the driver core.

In its standard or managed form, a device link combines both dependency types: It guarantees correct suspend/resume and shutdown ordering between a "supplier" device and its "consumer" devices, and it guarantees driver presence on the supplier. The consumer devices are not probed before the supplier is bound to a driver, and they're unbound before the supplier is unbound.

When driver presence on the supplier is irrelevant and only correct suspend/resume and shutdown ordering is needed, the device link may simply be set up with the `DL_FLAG_STATELESS` flag. In other words, enforcing driver presence on the supplier is optional.

Another optional feature is runtime PM integration: By setting the `DL_FLAG_PM_RUNTIME` flag on addition of the device link, the PM core is instructed to runtime resume the supplier and keep it active whenever and for as long as the consumer is runtime resumed.

11.1 Usage

The earliest point in time when device links can be added is after `device_add()` has been called for the supplier and `device_initialize()` has been called for the consumer.

It is legal to add them later, but care must be taken that the system remains in a consistent state: E.g. a device link cannot be added in the midst of a suspend/resume transition, so either commencement of such a transition needs to be prevented with `lock_system_sleep()`, or the device link needs to be added from a function which is guaranteed not to run in parallel to a suspend/resume transition, such as from a device `->probe` callback or a boot-time PCI quirk.

Another example for an inconsistent state would be a device link that represents a driver presence dependency, yet is added from the consumer's `->probe` callback while the supplier hasn't started to probe yet: Had the driver core known about the device link earlier, it wouldn't have probed the consumer in the first place. The onus is thus on the consumer to check presence of the supplier after adding the link, and defer probing on non-presence. [Note that it is valid to create a link from the consumer's `->probe` callback while the supplier is still probing, but the consumer must know that the supplier is functional already at the link creation time (that is the case, for instance, if the consumer has just acquired some resources that would not have been available had the supplier not been functional then).]

If a device link with `DL_FLAG_STATELESS` set (i.e. a stateless device link) is added in the `->probe` callback of the supplier or consumer driver, it is typically deleted in its `->remove` callback for symmetry. That way, if the driver is compiled as a module, the device link is added on module load and orderly deleted on unload. The same restrictions that apply to device link addition (e.g. exclusion of a parallel suspend/resume transition) apply equally to deletion. Device links managed by the driver core are deleted automatically by it.

Several flags may be specified on device link addition, two of which have already been mentioned above: `DL_FLAG_STATELESS` to express that no driver presence dependency is needed (but only correct suspend/resume and shutdown ordering) and `DL_FLAG_PM_RUNTIME` to express that runtime PM integration is desired.

Two other flags are specifically targeted at use cases where the device link is added from the consumer's `->probe` callback: `DL_FLAG_RPM_ACTIVE` can be specified to runtime resume the supplier and prevent it from suspending before the consumer is runtime suspended. `DL_FLAG_AUTOREMOVE_CONSUMER` causes the device link to be automatically purged when the consumer fails to probe or later unbinds.

Similarly, when the device link is added from supplier's `->probe` callback, `DL_FLAG_AUTOREMOVE_SUPPLIER` causes the device link to be automatically purged when the supplier fails to probe or later unbinds.

If neither `DL_FLAG_AUTOREMOVE_CONSUMER` nor `DL_FLAG_AUTOREMOVE_SUPPLIER` is set, `DL_FLAG_AUTOPROBE_CONSUMER` can be used to request the driver core to probe for a driver for the consumer driver on the link automatically after a driver has been bound to the supplier device.

Note, however, that any combinations of `DL_FLAG_AUTOREMOVE_CONSUMER`, `DL_FLAG_AUTOREMOVE_SUPPLIER` or `DL_FLAG_AUTOPROBE_CONSUMER` with

DL_FLAG_STATELESS are invalid and cannot be used.

11.2 Limitations

Driver authors should be aware that a driver presence dependency for managed device links (i.e. when DL_FLAG_STATELESS is not specified on link addition) may cause probing of the consumer to be deferred indefinitely. This can become a problem if the consumer is required to probe before a certain initcall level is reached. Worse, if the supplier driver is blacklisted or missing, the consumer will never be probed.

Moreover, managed device links cannot be deleted directly. They are deleted by the driver core when they are not necessary any more in accordance with the DL_FLAG_AUTOREMOVE_CONSUMER and DL_FLAG_AUTOREMOVE_SUPPLIER flags. However, stateless device links (i.e. device links with DL_FLAG_STATELESS set) are expected to be removed by whoever called device_link_add() to add them with the help of either device_link_del() or device_link_remove().

Passing DL_FLAG_RPM_ACTIVE along with DL_FLAG_STATELESS to device_link_add() may cause the PM-runtime usage counter of the supplier device to remain nonzero after a subsequent invocation of either device_link_del() or device_link_remove() to remove the device link returned by it. This happens if device_link_add() is called twice in a row for the same consumer-supplier pair without removing the link between these calls, in which case allowing the PM-runtime usage counter of the supplier to drop on an attempt to remove the link may cause it to be suspended while the consumer is still PM-runtime-active and that has to be avoided. [To work around this limitation it is sufficient to let the consumer runtime suspend at least once, or call pm_runtime_set_suspended() for it with PM-runtime disabled, between the device_link_add() and device_link_del() or device_link_remove() calls.]

Sometimes drivers depend on optional resources. They are able to operate in a degraded mode (reduced feature set or performance) when those resources are not present. An example is an SPI controller that can use a DMA engine or work in PIO mode. The controller can determine presence of the optional resources at probe time but on non-presence there is no way to know whether they will become available in the near future (due to a supplier driver probing) or never. Consequently it cannot be determined whether to defer probing or not. It would be possible to notify drivers when optional resources become available after probing, but it would come at a high cost for drivers as switching between modes of operation at runtime based on the availability of such resources would be much more complex than a mechanism based on probe deferral. In any case optional resources are beyond the scope of device links.

11.3 Examples

- An MMU device exists alongside a busmaster device, both are in the same power domain. The MMU implements DMA address translation for the busmaster device and shall be runtime resumed and kept active whenever and as long as the busmaster device is active. The busmaster device's driver shall not bind before the MMU is bound. To achieve this, a device link with runtime PM integration is added from the busmaster device (consumer) to the MMU device (supplier). The effect with regards to runtime PM is the same as if the MMU was the parent of the master device.

  The fact that both devices share the same power domain would normally suggest usage of a `struct dev_pm_domain` or `struct generic_pm_domain`, however these are not independent devices that happen to share a power switch, but rather the MMU device serves the busmaster device and is useless without it. A device link creates a synthetic hierarchical relationship between the devices and is thus more apt.

- A Thunderbolt host controller comprises a number of PCIe hotplug ports and an NHI device to manage the PCIe switch. On resume from system sleep, the NHI device needs to re-establish PCI tunnels to attached devices before the hotplug ports can resume. If the hotplug ports were children of the NHI, this resume order would automatically be enforced by the PM core, but unfortunately they're aunts. The solution is to add device links from the hotplug ports (consumers) to the NHI device (supplier). A driver presence dependency is not necessary for this use case.

- Discrete GPUs in hybrid graphics laptops often feature an HDA controller for HDMI/DP audio. In the device hierarchy the HDA controller is a sibling of the VGA device, yet both share the same power domain and the HDA controller is only ever needed when an HDMI/DP display is attached to the VGA device. A device link from the HDA controller (consumer) to the VGA device (supplier) aptly represents this relationship.

- ACPI allows definition of a device start order by way of _DEP objects. A classical example is when ACPI power management methods on one device are implemented in terms of I^2C accesses and require a specific I^2C controller to be present and functional for the power management of the device in question to work.

- In some SoCs a functional dependency exists from display, video codec and video processing IP cores on transparent memory access IP cores that handle burst access and compression/decompression.

11.4 Alternatives

- A `struct dev_pm_domain` can be used to override the bus, class or device type callbacks. It is intended for devices sharing a single on/off switch, however it does not guarantee a specific suspend/resume ordering, this needs to be implemented separately. It also does not by itself track the runtime PM status of the involved devices and turn off the power switch only when all of them are runtime suspended. Furthermore it cannot be used to enforce a specific shutdown ordering or a driver presence dependency.
- A `struct generic_pm_domain` is a lot more heavyweight than a device link and does not allow for shutdown ordering or driver presence dependencies. It also cannot be used on ACPI systems.

11.5 Implementation

The device hierarchy, which – as the name implies – is a tree, becomes a directed acyclic graph once device links are added.

Ordering of these devices during suspend/resume is determined by the dpm_list. During shutdown it is determined by the devices_kset. With no device links present, the two lists are a flattened, one-dimensional representations of the device tree such that a device is placed behind all its ancestors. That is achieved by traversing the ACPI namespace or OpenFirmware device tree top-down and appending devices to the lists as they are discovered.

Once device links are added, the lists need to satisfy the additional constraint that a device is placed behind all its suppliers, recursively. To ensure this, upon addition of the device link the consumer and the entire sub-graph below it (all children and consumers of the consumer) are moved to the end of the list. (Call to `device_reorder_to_tail()` from `device_link_add()`.)

To prevent introduction of dependency loops into the graph, it is verified upon device link addition that the supplier is not dependent on the consumer or any children or consumers of the consumer. (Call to `device_is_dependent()` from `device_link_add()`.) If that constraint is violated, `device_link_add()` will return `NULL` and a `WARNING` will be logged.

Notably this also prevents the addition of a device link from a parent device to a child. However the converse is allowed, i.e. a device link from a child to a parent. Since the driver core already guarantees correct suspend/resume and shutdown ordering between parent and child, such a device link only makes sense if a driver presence dependency is needed on top of that. In this case driver authors should weigh carefully if a device link is at all the right tool for the purpose. A more suitable approach might be to simply use deferred probing or add a device flag causing the parent driver to be probed before the child one.

11.6 State machine

enum **device_link_state**
Device link states.

Constants

DL_STATE_NONE The presence of the drivers is not being tracked.

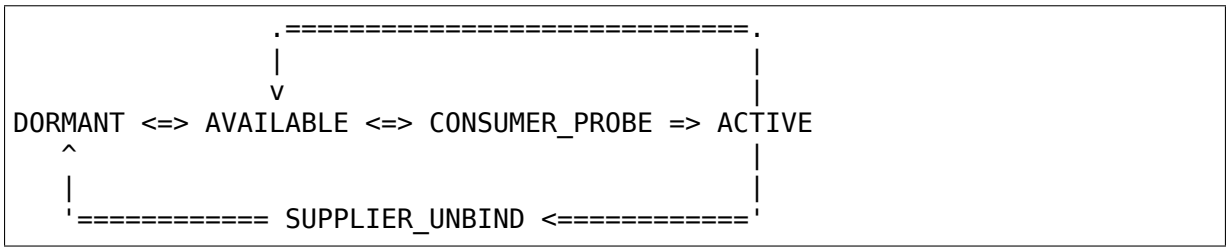
DL_STATE_DORMANT None of the supplier/consumer drivers is present.

DL_STATE_AVAILABLE The supplier driver is present, but the consumer is not.

DL_STATE_CONSUMER_PROBE The consumer is probing (supplier driver present).

DL_STATE_ACTIVE Both the supplier and consumer drivers are present.

DL_STATE_SUPPLIER_UNBIND The supplier driver is unbinding.



- The initial state of a device link is automatically determined by `device_link_add()` based on the driver presence on the supplier and consumer. If the link is created before any devices are probed, it is set to `DL_STATE_DORMANT`.
- When a supplier device is bound to a driver, links to its consumers progress to `DL_STATE_AVAILABLE`. (Call to `device_links_driver_bound()` from `driver_bound()`.)
- Before a consumer device is probed, presence of supplier drivers is verified by checking the consumer device is not in the wait_for_suppliers list and by checking that links to suppliers are in `DL_STATE_AVAILABLE` state. The state of the links is updated to `DL_STATE_CONSUMER_PROBE`. (Call to `device_links_check_suppliers()` from `really_probe()`.) This prevents the supplier from unbinding. (Call to `wait_for_device_probe()` from `device_links_unbind_consumers()`.)
- If the probe fails, links to suppliers revert back to `DL_STATE_AVAILABLE`. (Call to `device_links_no_driver()` from `really_probe()`.)
- If the probe succeeds, links to suppliers progress to `DL_STATE_ACTIVE`. (Call to `device_links_driver_bound()` from `driver_bound()`.)
- When the consumer's driver is later on removed, links to suppliers revert back to `DL_STATE_AVAILABLE`. (Call to `__device_links_no_driver()` from `device_links_driver_cleanup()`, which in turn is called from `__device_release_driver()`.)
- Before a supplier's driver is removed, links to consumers that are not bound to a driver are updated to `DL_STATE_SUPPLIER_UNBIND`. (Call to `device_links_busy()` from `__device_release_driver()`.) This prevents

the consumers from binding. (Call to `device_links_check_suppliers()` from `really_probe()`.) Consumers that are bound are freed from their driver; consumers that are probing are waited for until they are done. (Call to `device_links_unbind_consumers()` from `__device_release_driver()`.) Once all links to consumers are in `DL_STATE_SUPPLIER_UNBIND` state, the supplier driver is released and the links revert to `DL_STATE_DORMANT`. (Call to `device_links_driver_cleanup()` from `__device_release_driver()`.)

11.7 API

struct device_link * **device_link_add**(struct device * consumer, struct device * supplier, u32 flags)

Create a link between two devices.

Parameters

struct device * consumer Consumer end of the link.

struct device * supplier Supplier end of the link.

u32 flags Link flags.

Description

The caller is responsible for the proper synchronization of the link creation with runtime PM. First, setting the DL_FLAG_PM_RUNTIME flag will cause the runtime PM framework to take the link into account. Second, if the DL_FLAG_RPM_ACTIVE flag is set in addition to it, the supplier devices will be forced into the active metastate and reference-counted upon the creation of the link. If DL_FLAG_PM_RUNTIME is not set, DL_FLAG_RPM_ACTIVE will be ignored.

If DL_FLAG_STATELESS is set in **flags**, the caller of this function is expected to release the link returned by it directly with the help of either `device_link_del()` or `device_link_remove()`.

If that flag is not set, however, the caller of this function is handing the management of the link over to the driver core entirely and its return value can only be used to check whether or not the link is present. In that case, the DL_FLAG_AUTOREMOVE_CONSUMER and DL_FLAG_AUTOREMOVE_SUPPLIER device link flags can be used to indicate to the driver core when the link can be safely deleted. Namely, setting one of them in **flags** indicates to the driver core that the link is not going to be used (by the given caller of this function) after unbinding the consumer or supplier driver, respectively, from its device, so the link can be deleted at that point. If none of them is set, the link will be maintained until one of the devices pointed to by it (either the consumer or the supplier) is unregistered.

Also, if DL_FLAG_STATELESS, DL_FLAG_AUTOREMOVE_CONSUMER and DL_FLAG_AUTOREMOVE_SUPPLIER are not set in **flags** (that is, a persistent managed device link is being added), the DL_FLAG_AUTOPROBE_CONSUMER flag can be used to request the driver core to automaticall probe for a consmer driver after successfully binding a driver to the supplier device.

The combination of DL_FLAG_STATELESS and one of DL_FLAG_AUTOREMOVE_CONSUMER, DL_FLAG_AUTOREMOVE_SUPPLIER, or DL_FLAG_AUTOPROBE_CONSUMER set in **flags** at the same time is invalid and will cause NULL to be returned upfront. However, if a device link between the given **consumer** and **supplier** pair exists already when this function is called for them, the existing link will be returned regardless of its current type and status (the link's flags may be modified then). The caller of this function is then expected to treat the link as though it has just been created, so (in particular) if DL_FLAG_STATELESS was passed in **flags**, the link needs to be released explicitly when not needed any more (as stated above).

A side effect of the link creation is re-ordering of dpm_list and the devices_kset list by moving the consumer device and all devices depending on it to the ends of these lists (that does not happen to devices that have not been registered when this function is called).

The supplier device is required to be registered when this function is called and NULL will be returned if that is not the case. The consumer device need not be registered, however.

void **device_link_del**(struct device_link * link)
Delete a stateless link between two devices.

Parameters

struct device_link * link Device link to delete.

Description

The caller must ensure proper synchronization of this function with runtime PM. If the link was added multiple times, it needs to be deleted as often. Care is required for hotplugged devices: Their links are purged on removal and calling `device_link_del()` is then no longer allowed.

void **device_link_remove**(void * consumer, struct device * supplier)
Delete a stateless link between two devices.

Parameters

void * consumer Consumer end of the link.

struct device * supplier Supplier end of the link.

Description

The caller must ensure proper synchronization of this function with runtime PM.

CHAPTER

TWELVE

COMPONENT HELPER FOR AGGREGATE DRIVERS

The component helper allows drivers to collect a pile of sub-devices, including their bound drivers, into an aggregate driver. Various subsystems already provide functions to get hold of such components, e.g. of_clk_get_by_name(). The component helper can be used when such a subsystem-specific way to find a device is not available: The component helper fills the niche of aggregate drivers for specific hardware, where further standardization into a subsystem would not be practical. The common example is when a logical device (e.g. a DRM display driver) is spread around the SoC on various components (scanout engines, blending blocks, transcoders for various outputs and so on).

The component helper also doesn't solve runtime dependencies, e.g. for system suspend and resume operations. See also device links.

Components are registered using `component_add()` and unregistered with `component_del()`, usually from the driver's probe and disconnect functions.

Aggregate drivers first assemble a component match list of what they need using `component_match_add()`. This is then registered as an aggregate driver using `component_master_add_with_match()`, and unregistered using `component_master_del()`.

12.1 API

struct **component_ops**
callbacks for component drivers

Definition

```
struct component_ops {
  int (*bind)(struct device *comp, struct device *master, void *master_
↪data);
  void (*unbind)(struct device *comp, struct device *master, void *master_
↪data);
};
```

Members

bind Called through `component_bind_all()` when the aggregate driver is ready to bind the overall driver.

unbind Called through `component_unbind_all()` when the aggregate driver is ready to bind the overall driver, or when `component_bind_all()` fails part-ways through and needs to unbind some already bound components.

Description

Components are registered with `component_add()` and unregistered with `component_del()`.

struct **component_master_ops**
callback for the aggregate driver

Definition

```
struct component_master_ops {
  int (*bind)(struct device *master);
  void (*unbind)(struct device *master);
};
```

Members

bind Called when all components or the aggregate driver, as specified in the match list passed to `component_master_add_with_match()`, are ready. Usually there are 3 steps to bind an aggregate driver:

1. Allocate a structure for the aggregate driver.
2. Bind all components to the aggregate driver by calling `component_bind_all()` with the aggregate driver structure as opaque pointer data.
3. Register the aggregate driver with the subsystem to publish its interfaces.

Note that the lifetime of the aggregate driver does not align with any of the underlying `struct device` instances. Therefore devm cannot be used and all resources acquired or allocated in this callback must be explicitly released in the **unbind** callback.

unbind Called when either the aggregate driver, using `component_master_del()`, or one of its components, using `component_del()`, is unregistered.

Description

Aggregate drivers are registered with `component_master_add_with_match()` and unregistered with `component_master_del()`.

void **component_match_add**(struct device * master, struct component_match ** matchptr, int (*compare)(struct device *, void *), void * compare_data)
add a component match entry

Parameters

struct device * master device with the aggregate driver

struct component_match ** matchptr pointer to the list of component matches

int (*)(struct device *, void *) compare compare function to match against all components

void * compare_data opaque pointer passed to the **compare** function

Description

Adds a new component match to the list stored in **matchptr**, which the **master** aggregate driver needs to function. The list of component matches pointed to by **matchptr** must be initialized to NULL before adding the first match. This only matches against components added with `component_add()`.

The allocated match list in **matchptr** is automatically released using devm actions.

See also `component_match_add_release()` and `component_match_add_typed()`.

void **component_match_add_release**(struct device * master, struct component_match ** matchptr, void (*release)(struct device *, void *), int (*compare) (struct device *, void *), void * compare_data)

add a component match entry with release callback

Parameters

struct device * master device with the aggregate driver

struct component_match ** matchptr pointer to the list of component matches

void (*)(struct device *, void *) release release function for **compare_data**

int (*)(struct device *, void *) compare compare function to match against all components

void * compare_data opaque pointer passed to the **compare** function

Description

Adds a new component match to the list stored in **matchptr**, which the **master** aggregate driver needs to function. The list of component matches pointed to by **matchptr** must be initialized to NULL before adding the first match. This only matches against components added with `component_add()`.

The allocated match list in **matchptr** is automatically released using devm actions, where upon **release** will be called to free any references held by **compare_data**, e.g. when **compare_data** is a `device_node` that must be released with of_node_put().

See also `component_match_add()` and `component_match_add_typed()`.

void **component_match_add_typed**(struct device * master, struct component_match ** matchptr, int (*compare_typed)(struct device *, int, void *), void * compare_data)

add a component match entry for a typed component

Parameters

struct device * master device with the aggregate driver

struct component_match ** matchptr pointer to the list of component matches

int (*)(struct device *, int, void *) compare_typed compare function to match against all typed components

void * compare_data opaque pointer passed to the **compare** function

Description

Adds a new component match to the list stored in **matchptr**, which the **master** aggregate driver needs to function. The list of component matches pointed to by **matchptr** must be initialized to NULL before adding the first match. This only matches against components added with `component_add_typed()`.

The allocated match list in **matchptr** is automatically released using devm actions.

See also `component_match_add_release()` and `component_match_add_typed()`.

int **component_master_add_with_match**(struct device * dev, const struct component_master_ops * ops, struct component_match * match)
register an aggregate driver

Parameters

struct device * dev device with the aggregate driver

const struct component_master_ops * ops callbacks for the aggregate driver

struct component_match * match component match list for the aggregate driver

Description

Registers a new aggregate driver consisting of the components added to **match** by calling one of the `component_match_add()` functions. Once all components in **match** are available, it will be assembled by calling `component_master_ops.bind` from **ops**. Must be unregistered by calling `component_master_del()`.

void **component_master_del**(struct device * dev, const struct component_master_ops * ops)
unregister an aggregate driver

Parameters

struct device * dev device with the aggregate driver

const struct component_master_ops * ops callbacks for the aggregate driver

Description

Unregisters an aggregate driver registered with `component_master_add_with_match()`. If necessary the aggregate driver is first disassembled by calling `component_master_ops.unbind` from **ops**.

void **component_unbind_all**(struct device * master_dev, void * data)
unbind all components of an aggregate driver

Parameters

struct device * master_dev device with the aggregate driver

void * data opaque pointer, passed to all components

Description

Unbinds all components of the aggregate **dev** by passing **data** to their `component_ops.unbind` functions. Should be called from `component_master_ops.unbind`.

int **component_bind_all**(struct device * master_dev, void * data)
bind all components of an aggregate driver

Parameters

struct device * master_dev device with the aggregate driver

void * data opaque pointer, passed to all components

Description

Binds all components of the aggregate **dev** by passing **data** to their `component_ops.bind` functions. Should be called from `component_master_ops.bind`.

int **component_add_typed**(struct device * dev, const struct component_ops * ops, int subcomponent)
register a component

Parameters

struct device * dev component device

const struct component_ops * ops component callbacks

int subcomponent nonzero identifier for subcomponents

Description

Register a new component for **dev**. Functions in **ops** will be call when the aggregate driver is ready to bind the overall driver by calling `component_bind_all()`. See also `struct component_ops`.

subcomponent must be nonzero and is used to differentiate between multiple components registerd on the same device **dev**. These components are match using `component_match_add_typed()`.

The component needs to be unregistered at driver unload/disconnect by calling `component_del()`.

See also `component_add()`.

int **component_add**(struct device * dev, const struct component_ops * ops)
register a component

Parameters

struct device * dev component device

const struct component_ops * ops component callbacks

Description

Register a new component for **dev**. Functions in **ops** will be called when the aggregate driver is ready to bind the overall driver by calling `component_bind_all()`. See also `struct component_ops`.

The component needs to be unregistered at driver unload/disconnect by calling `component_del()`.

See also `component_add_typed()` for a variant that allows multipled different components on the same device.

void **component_del**(struct device * dev, const struct component_ops * ops)
unregister a component

Parameters

struct device * dev component device

const struct component_ops * ops component callbacks

Description

Unregister a component added with `component_add()`. If the component is bound into an aggregate driver, this will force the entire aggregate driver, including all its components, to be unbound.

CHAPTER

THIRTEEN

MESSAGE-BASED DEVICES

13.1 Fusion message devices

u8 **mpt_register**(MPT_CALLBACK cbfunc, MPT_DRIVER_CLASS dclass, char * func_name)
Register protocol-specific main callback handler.

Parameters

MPT_CALLBACK cbfunc callback function pointer

MPT_DRIVER_CLASS dclass Protocol driver's class (MPT_DRIVER_CLASS enum value)

char * func_name call function's name

This routine is called by a protocol-specific driver (SCSI host, LAN, SCSI target) to register its reply callback routine. Each protocol-specific driver must do this before it will be able to use any IOC resources, such as obtaining request frames.

NOTES

The SCSI protocol driver currently calls this routine thrice in order to register separate callbacks; one for "normal" SCSI IO; one for MptScsiTaskMgmt requests; one for Scan/DV requests.

Returns u8 valued "handle" in the range (and S.O.D. order) {N,⋯,7,6,5,⋯,1} if successful. A return value of MPT_MAX_PROTOCOL_DRIVERS (including zero!) should be considered an error by the caller.

void **mpt_deregister**(u8 cb_idx)
Deregister a protocol drivers resources.

Parameters

u8 cb_idx previously registered callback handle

Each protocol-specific driver should call this routine when its module is unloaded.

int **mpt_event_register**(u8 cb_idx, MPT_EVHANDLER ev_cbfunc)
Register protocol-specific event callback handler.

Parameters

u8 cb_idx previously registered (via mpt_register) callback handle

MPT_EVHANDLER ev_cbfunc callback function

This routine can be called by one or more protocol-specific drivers if/when they choose to be notified of MPT events.

Returns 0 for success.

void **mpt_event_deregister**(u8 cb_idx)
Deregister protocol-specific event callback handler

Parameters

u8 cb_idx previously registered callback handle

Each protocol-specific driver should call this routine when it does not (or can no longer) handle events, or when its module is unloaded.

int **mpt_reset_register**(u8 cb_idx, MPT_RESETHANDLER reset_func)
Register protocol-specific IOC reset handler.

Parameters

u8 cb_idx previously registered (via mpt_register) callback handle

MPT_RESETHANDLER reset_func reset function

This routine can be called by one or more protocol-specific drivers if/when they choose to be notified of IOC resets.

Returns 0 for success.

void **mpt_reset_deregister**(u8 cb_idx)
Deregister protocol-specific IOC reset handler.

Parameters

u8 cb_idx previously registered callback handle

Each protocol-specific driver should call this routine when it does not (or can no longer) handle IOC reset handling, or when its module is unloaded.

int **mpt_device_driver_register**(struct mpt_pci_driver * dd_cbfunc, u8 cb_idx)
Register device driver hooks

Parameters

struct mpt_pci_driver * dd_cbfunc driver callbacks struct

u8 cb_idx MPT protocol driver index

void **mpt_device_driver_deregister**(u8 cb_idx)
DeRegister device driver hooks

Parameters

u8 cb_idx MPT protocol driver index

MPT_FRAME_HDR* **mpt_get_msg_frame**(u8 cb_idx, MPT_ADAPTER * ioc)
Obtain an MPT request frame from the pool

Parameters

u8 cb_idx Handle of registered MPT protocol driver

MPT_ADAPTER * ioc Pointer to MPT adapter structure

Obtain an MPT request frame from the pool (of 1024) that are allocated per MPT adapter.

Returns pointer to a MPT request frame or NULL if none are available or IOC is not active.

void **mpt_put_msg_frame**(u8 cb_idx, MPT_ADAPTER * ioc, MPT_FRAME_HDR * mf)

Send a protocol-specific MPT request frame to an IOC

Parameters

u8 cb_idx Handle of registered MPT protocol driver

MPT_ADAPTER * ioc Pointer to MPT adapter structure

MPT_FRAME_HDR * mf Pointer to MPT request frame

This routine posts an MPT request frame to the request post FIFO of a specific MPT adapter.

void **mpt_put_msg_frame_hi_pri**(u8 cb_idx, MPT_ADAPTER * ioc, MPT_FRAME_HDR * mf)

Send a hi-pri protocol-specific MPT request frame

Parameters

u8 cb_idx Handle of registered MPT protocol driver

MPT_ADAPTER * ioc Pointer to MPT adapter structure

MPT_FRAME_HDR * mf Pointer to MPT request frame

Send a protocol-specific MPT request frame to an IOC using hi-priority request queue.

This routine posts an MPT request frame to the request post FIFO of a specific MPT adapter.

void **mpt_free_msg_frame**(MPT_ADAPTER * ioc, MPT_FRAME_HDR * mf)

Place MPT request frame back on FreeQ.

Parameters

MPT_ADAPTER * ioc Pointer to MPT adapter structure

MPT_FRAME_HDR * mf Pointer to MPT request frame

This routine places a MPT request frame back on the MPT adapter's FreeQ.

int **mpt_send_handshake_request**(u8 cb_idx, MPT_ADAPTER * ioc, int reqBytes, u32 * req, int sleepFlag)

Send MPT request via doorbell handshake method.

Parameters

u8 cb_idx Handle of registered MPT protocol driver

MPT_ADAPTER * ioc Pointer to MPT adapter structure

int reqBytes Size of the request in bytes

u32 * req Pointer to MPT request frame

int sleepFlag Use schedule if CAN_SLEEP else use udelay.

This routine is used exclusively to send MptScsiTaskMgmt requests since they are required to be sent via doorbell handshake.

NOTE

It is the callers responsibility to byte-swap fields in the request which are greater than 1 byte in size.

Returns 0 for success, non-zero for failure.

int **mpt_verify_adapter**(int iocid, MPT_ADAPTER ** iocpp)
Given IOC identifier, set pointer to its adapter structure.

Parameters

int iocid IOC unique identifier (integer)

MPT_ADAPTER ** iocpp Pointer to pointer to IOC adapter

Given a unique IOC identifier, set pointer to the associated MPT adapter structure.

Returns iocid and sets iocpp if iocid is found. Returns -1 if iocid is not found.

int **mpt_attach**(struct pci_dev * pdev, const struct pci_device_id * id)
Install a PCI intelligent MPT adapter.

Parameters

struct pci_dev * pdev Pointer to pci_dev structure

const struct pci_device_id * id PCI device ID information

This routine performs all the steps necessary to bring the IOC of a MPT adapter to a OPERATIONAL state. This includes registering memory regions, registering the interrupt, and allocating request and reply memory pools.

This routine also pre-fetches the LAN MAC address of a Fibre Channel MPT adapter.

Returns 0 for success, non-zero for failure.

TODO: Add support for polled controllers

void **mpt_detach**(struct pci_dev * pdev)
Remove a PCI intelligent MPT adapter.

Parameters

struct pci_dev * pdev Pointer to pci_dev structure

int **mpt_suspend**(struct pci_dev * pdev, pm_message_t state)
Fusion MPT base driver suspend routine.

Parameters

struct pci_dev * pdev Pointer to pci_dev structure

pm_message_t state new state to enter

int **mpt_resume**(struct pci_dev * pdev)
Fusion MPT base driver resume routine.

Parameters

struct pci_dev * pdev Pointer to pci_dev structure

u32 **mpt_GetIocState**(MPT_ADAPTER * ioc, int cooked)
Get the current state of a MPT adapter.

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int cooked Request raw or cooked IOC state

Returns all IOC Doorbell register bits if cooked==0, else just the Doorbell bits in MPI_IOC_STATE_MASK.

int **mpt_alloc_fw_memory**(MPT_ADAPTER * ioc, int size)
allocate firmware memory

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int size total FW bytes

If memory has already been allocated, the same (cached) value is returned.

Return 0 if successful, or non-zero for failure

void **mpt_free_fw_memory**(MPT_ADAPTER * ioc)
free firmware memory

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

If alt_img is NULL, delete from ioc structure. Else, delete a secondary image in same format.

int **mptbase_sas_persist_operation**(MPT_ADAPTER * ioc, u8 persist_opcode)
Perform operation on SAS Persistent Table

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

u8 persist_opcode see below

MPI_SAS_OP_CLEAR_NOT_PRESENT	Free all persist TargetID mappings for devices not currently present.
MPI_SAS_OP_CLEAR_ALL_PERSISTENT	Clear all persist TargetID mappings

NOTE

Don’t use not this function during interrupt time.

Returns 0 for success, non-zero error

int **mpt_raid_phys_disk_pg0**(MPT_ADAPTER * ioc, u8 phys_disk_num, RaidPhysDiskPage0_t * phys_disk)
returns phys disk page zero

Parameters

MPT_ADAPTER * ioc Pointer to a Adapter Structure

u8 phys_disk_num io unit unique phys disk num generated by the ioc

RaidPhysDiskPage0_t * phys_disk requested payload data returned

Return

0 on success -EFAULT if read of config page header fails or data pointer not NULL -ENOMEM if pci_alloc failed

int **mpt_raid_phys_disk_get_num_paths**(MPT_ADAPTER * ioc, u8 phys_disk_num)
returns number paths associated to this phys_num

Parameters

MPT_ADAPTER * ioc Pointer to a Adapter Structure

u8 phys_disk_num io unit unique phys disk num generated by the ioc

Return

returns number paths

int **mpt_raid_phys_disk_pg1**(MPT_ADAPTER * ioc, u8 phys_disk_num, RaidPhysDiskPage1_t * phys_disk)
returns phys disk page 1

Parameters

MPT_ADAPTER * ioc Pointer to a Adapter Structure

u8 phys_disk_num io unit unique phys disk num generated by the ioc

RaidPhysDiskPage1_t * phys_disk requested payload data returned

Return

0 on success -EFAULT if read of config page header fails or data pointer not NULL -ENOMEM if pci_alloc failed

int **mpt_findImVolumes**(MPT_ADAPTER * ioc)
Identify IDs of hidden disks and RAID Volumes

Parameters

MPT_ADAPTER * ioc Pointer to a Adapter Strucutre

Return

0 on success -EFAULT if read of config page header fails or data pointer not NULL -ENOMEM if pci_alloc failed

int **mpt_config**(MPT_ADAPTER * ioc, CONFIGPARMS * pCfg)
Generic function to issue config message

Parameters

MPT_ADAPTER * ioc Pointer to an adapter structure

CONFIGPARMS * pCfg Pointer to a configuration structure. Struct contains action, page address, direction, physical address and pointer to a configuration page header Page header is updated.

Returns 0 for success -EPERM if not allowed due to ISR context -EAGAIN if no msg frames currently available -EFAULT for non-successful reply or no reply (timeout)

void **mpt_print_ioc_summary**(MPT_ADAPTER * ioc, char * buffer, int * size, int len, int showlan)
Write ASCII summary of IOC to a buffer.

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

char * buffer Pointer to buffer where IOC summary info should be written

int * size Pointer to number of bytes we wrote (set by this routine)

int len Offset at which to start writing in buffer

int showlan Display LAN stuff?

This routine writes (english readable) ASCII text, which represents a summary of IOC information, to a buffer.

int **mpt_set_taskmgmt_in_progress_flag**(MPT_ADAPTER * ioc)
set flags associated with task management

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

Returns 0 for SUCCESS or -1 if FAILED.

If -1 is return, then it was not possible to set the flags

void **mpt_clear_taskmgmt_in_progress_flag**(MPT_ADAPTER * ioc)
clear flags associated with task management

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

void **mpt_halt_firmware**(MPT_ADAPTER * ioc)
Halts the firmware if it is operational and panic the kernel

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int **mpt_Soft_Hard_ResetHandler**(MPT_ADAPTER * ioc, int sleepFlag)
Try less expensive reset

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int sleepFlag Indicates if sleep or schedule must be called.

Returns 0 for SUCCESS or -1 if FAILED. Try for softreset first, only if it fails go for expensive HardReset.

int **mpt_HardResetHandler**(MPT_ADAPTER * ioc, int sleepFlag)
Generic reset handler

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int sleepFlag Indicates if sleep or schedule must be called.

Issues SCSI Task Management call based on input arg values. If TaskMgmt fails, returns associated SCSI request.

Remark: _HardResetHandler can be invoked from an interrupt thread (timer) or a non-interrupt thread. In the former, must not call schedule().

Note

A return of -1 is a FATAL error case, as it means a FW reload/initialization failed.

Returns 0 for SUCCESS or -1 if FAILED.

const char * **mptscsih_info**(struct Scsi_Host * SChost)
Return information about MPT adapter

Parameters

struct Scsi_Host * SChost Pointer to Scsi_Host structure

(linux scsi_host_template.info routine)

Returns pointer to buffer where information was written.

int **mptscsih_qcmd**(struct scsi_cmnd * SCpnt)
Primary Fusion MPT SCSI initiator IO start routine.

Parameters

struct scsi_cmnd * SCpnt Pointer to scsi_cmnd structure

(linux scsi_host_template.queuecommand routine) This is the primary SCSI IO start routine. Create a MPI SCSIIORequest from a linux scsi_cmnd request and send it to the IOC.

Returns 0. (rtn value discarded by linux scsi mid-layer)

int **mptscsih_IssueTaskMgmt**(MPT_SCSI_HOST * hd, u8 type, u8 channel,
u8 id, u64 lun, int ctx2abort, ulong timeout)
Generic send Task Management function.

Parameters

MPT_SCSI_HOST * hd Pointer to MPT_SCSI_HOST structure

u8 type Task Management type

u8 channel channel number for task management

u8 id Logical Target ID for reset (if appropriate)

u64 lun Logical Unit for reset (if appropriate)

int ctx2abort Context for the task to be aborted (if appropriate)

ulong timeout timeout for task management control

Remark: _HardResetHandler can be invoked from an interrupt thread (timer) or a non-interrupt thread. In the former, must not call schedule().

Not all fields are meaningfull for all task types.

Returns 0 for SUCCESS, or FAILED.

int **mptscsih_abort**(struct scsi_cmnd * SCpnt)
Abort linux scsi_cmnd routine, new_eh variant

Parameters

struct scsi_cmnd * SCpnt Pointer to scsi_cmnd structure, IO to be aborted

(linux scsi_host_template.eh_abort_handler routine)

Returns SUCCESS or FAILED.

int **mptscsih_dev_reset**(struct scsi_cmnd * SCpnt)
Perform a SCSI TARGET_RESET! new_eh variant

Parameters

struct scsi_cmnd * SCpnt Pointer to scsi_cmnd structure, IO which reset is due to

(linux scsi_host_template.eh_dev_reset_handler routine)

Returns SUCCESS or FAILED.

int **mptscsih_bus_reset**(struct scsi_cmnd * SCpnt)
Perform a SCSI BUS_RESET! new_eh variant

Parameters

struct scsi_cmnd * SCpnt Pointer to scsi_cmnd structure, IO which reset is due to

(linux scsi_host_template.eh_bus_reset_handler routine)

Returns SUCCESS or FAILED.

int **mptscsih_host_reset**(struct scsi_cmnd * SCpnt)
Perform a SCSI host adapter RESET (new_eh variant)

Parameters

struct scsi_cmnd * SCpnt Pointer to scsi_cmnd structure, IO which reset is due to

(linux scsi_host_template.eh_host_reset_handler routine)

Returns SUCCESS or FAILED.

int **mptscsih_taskmgmt_complete**(MPT_ADAPTER * ioc, MPT_FRAME_HDR * mf, MPT_FRAME_HDR * mr)
Registered with Fusion MPT base driver

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

MPT_FRAME_HDR * mf Pointer to SCSI task mgmt request frame

MPT_FRAME_HDR * mr Pointer to SCSI task mgmt reply frame

This routine is called from mptbase.c::mpt_interrupt() at the completion of any SCSI task management request. This routine is registered with the MPT (base) driver at driver load/init time via the `mpt_register()` API call.

Returns 1 indicating alloc'd request frame ptr should be freed.

struct scsi_cmnd * **mptscsih_get_scsi_lookup**(MPT_ADAPTER * ioc, int i)
retrieves scmd entry

Parameters

MPT_ADAPTER * ioc Pointer to MPT_ADAPTER structure

int i index into the array

Description

Returns the scsi_cmd pointer

CHAPTER

FOURTEEN

INFINIBAND AND REMOTE DMA (RDMA) INTERFACES

14.1 Introduction and Overview

TBD

14.2 InfiniBand core interfaces

struct iwpm_nlmsg_request * **iwpm_get_nlmsg_request**(__u32 nlmsg_seq,
u8 nl_client,
gfp_t gfp)

Allocate and initialize netlink message request

Parameters

__u32 nlmsg_seq Sequence number of the netlink message

u8 nl_client The index of the netlink client

gfp_t gfp Indicates how the memory for the request should be allocated

Description

Returns the newly allocated netlink request object if successful, otherwise returns NULL

void **iwpm_free_nlmsg_request**(struct kref * kref)

Deallocate netlink message request

Parameters

struct kref * kref Holds reference of netlink message request

struct iwpm_nlmsg_request * **iwpm_find_nlmsg_request**(__u32 echo_seq)

Find netlink message request in the request list

Parameters

__u32 echo_seq Sequence number of the netlink request to find

Description

Returns the found netlink message request, if not found, returns NULL

int **iwpm_wait_complete_req**(struct iwpm_nlmsg_request * nlmsg_request)

Block while servicing the netlink request

Parameters

struct iwpm_nlmsg_request * nlmsg_request Netlink message request to service

Description

Wakes up, after the request is completed or expired Returns 0 if the request is complete without error

int **iwpm_get_nlmsg_seq**(void)
Get the sequence number for a netlink message to send to the port mapper

Parameters

void no arguments

Description

Returns the sequence number for the netlink message.

void **iwpm_add_remote_info**(struct iwpm_remote_info * reminfo)
Add remote address info of the connecting peer to the remote info hash table

Parameters

struct iwpm_remote_info * reminfo The remote info to be added

int **iwpm_valid_client**(u8 nl_client)
Check if the port mapper client is valid

Parameters

u8 nl_client The index of the netlink client

Description

Valid clients need to call iwpm_init() before using the port mapper

void **iwpm_set_valid**(u8 nl_client, int valid)
Set the port mapper client to valid or not

Parameters

u8 nl_client The index of the netlink client

int valid 1 if valid or 0 if invalid

u32 **iwpm_check_registration**(u8 nl_client, u32 reg)
Check if the client registration matches the given one

Parameters

u8 nl_client The index of the netlink client

u32 reg The given registration type to compare with

Description

Call iwpm_register_pid() to register a client Returns true if the client registration matches reg, otherwise returns false

void **iwpm_set_registration**(u8 nl_client, u32 reg)
Set the client registration

Parameters

u8 nl_client The index of the netlink client

u32 reg Registration type to set

u32 **iwpm_get_registration**(u8 nl_client)

Parameters

u8 nl_client The index of the netlink client

Description

Returns the client registration type

int **iwpm_send_mapinfo**(u8 nl_client, int iwpm_pid)
Send local and mapped IPv4/IPv6 address info of a client to the user space port mapper

Parameters

u8 nl_client The index of the netlink client

int iwpm_pid The pid of the user space port mapper

Description

If successful, returns the number of sent mapping info records

int **iwpm_mapinfo_available**(void)
Check if any mapping info records is available in the hash table

Parameters

void no arguments

Description

Returns 1 if mapping information is available, otherwise returns 0

int **iwpm_compare_sockaddr**(struct sockaddr_storage * a_sockaddr, struct sockaddr_storage * b_sockaddr)
Compare two sockaddr storage structs

Parameters

struct sockaddr_storage * a_sockaddr first sockaddr to compare

struct sockaddr_storage * b_sockaddr second sockaddr to compare

Return

0 if they are holding the same ip/tcp address info, otherwise returns 1

int **iwpm_validate_nlmsg_attr**(struct nlattr * nltb, int nla_count)
Check for NULL netlink attributes

Parameters

struct nlattr * nltb Holds address of each netlink message attributes

int nla_count Number of netlink message attributes

Description

Returns error if any of the nla_count attributes is NULL

struct sk_buff * **iwpm_create_nlmsg**(u32 nl_op, struct nlmsghdr ** nlh, int nl_client)
Allocate skb and form a netlink message

Parameters

u32 nl_op Netlink message opcode

struct nlmsghdr ** nlh Holds address of the netlink message header in skb

int nl_client The index of the netlink client

Description

Returns the newly allcated skb, or NULL if the tailroom of the skb is insufficient to store the message header and payload

int **iwpm_parse_nlmsg**(struct netlink_callback * cb, int policy_max, const struct nla_policy * nlmsg_policy, struct nlattr * nltb, const char * msg_type)
Validate and parse the received netlink message

Parameters

struct netlink_callback * cb Netlink callback structure

int policy_max Maximum attribute type to be expected

const struct nla_policy * nlmsg_policy Validation policy

struct nlattr * nltb Array to store policy_max parsed elements

const char * msg_type Type of netlink message

Description

Returns 0 on success or a negative error code

void **iwpm_print_sockaddr**(struct sockaddr_storage * sockaddr, char * msg)
Print IPv4/IPv6 address and TCP port

Parameters

struct sockaddr_storage * sockaddr Socket address to print

char * msg Message to print

int **iwpm_send_hello**(u8 nl_client, int iwpm_pid, u16 abi_version)
Send hello response to iwpmd

Parameters

u8 nl_client The index of the netlink client

int iwpm_pid The pid of the user space port mapper

u16 abi_version The kernel's abi_version

Description

Returns 0 on success or a negative error code

int **ib_process_cq_direct**(struct ib_cq * cq, int budget)
process a CQ in caller context

Parameters

struct ib_cq * cq CQ to process

int budget number of CQEs to poll for

Description

This function is used to process all outstanding CQ entries. It does not offload CQ processing to a different context and does not ask for completion interrupts from the HCA. Using direct processing on CQ with non IB_POLL_DIRECT type may trigger concurrent processing.

Note

do not pass -1 as `budget` unless it is guaranteed that the number of completions that will be processed is small.

struct ib_cq * **__ib_alloc_cq_user**(struct ib_device * dev, void * private, int nr_cqe, int comp_vector, enum ib_poll_context poll_ctx, const char * caller, struct ib_udata * udata)
allocate a completion queue

Parameters

struct ib_device * dev device to allocate the CQ for

void * private driver private data, accessible from cq->cq_context

int nr_cqe number of CQEs to allocate

int comp_vector HCA completion vectors for this CQ

enum ib_poll_context poll_ctx context to poll the CQ from.

const char * caller module owner name.

struct ib_udata * udata Valid user data or NULL for kernel object

Description

This is the proper interface to allocate a CQ for in-kernel users. A CQ allocated with this interface will automatically be polled from the specified context. The ULP must use wr->wr_cqe instead of wr->wr_id to use this CQ abstraction.

struct ib_cq * **__ib_alloc_cq_any**(struct ib_device * dev, void * private, int nr_cqe, enum ib_poll_context poll_ctx, const char * caller)
allocate a completion queue

Parameters

struct ib_device * dev device to allocate the CQ for

void * private driver private data, accessible from cq->cq_context

int nr_cqe number of CQEs to allocate

enum ib_poll_context poll_ctx context to poll the CQ from

const char * caller module owner name

Description

Attempt to spread ULP Completion Queues over each device's interrupt vectors. A simple best-effort mechanism is used.

void **ib_free_cq_user**(struct ib_cq * cq, struct ib_udata * udata)
free a completion queue

Parameters

struct ib_cq * cq completion queue to free.

struct ib_udata * udata User data or NULL for kernel object

struct ib_cq * **ib_cq_pool_get**(struct ib_device * dev, unsigned int nr_cqe, int comp_vector_hint, enum ib_poll_context poll_ctx)
Find the least used completion queue that matches a given cpu hint (or least used for wild card affinity) and fits nr_cqe.

Parameters

struct ib_device * dev rdma device

unsigned int nr_cqe number of needed cqe entries

int comp_vector_hint completion vector hint (-1) for the driver to assign a comp vector based on internal counter

enum ib_poll_context poll_ctx cq polling context

Description

Finds a cq that satisfies **comp_vector_hint** and **nr_cqe** requirements and claim entries in it for us. In case there is no available cq, allocate a new cq with the requirements and add it to the device pool. IB_POLL_DIRECT cannot be used for shared cqs so it is not a valid value for **poll_ctx**.

void **ib_cq_pool_put**(struct ib_cq * cq, unsigned int nr_cqe)
Return a CQ taken from a shared pool.

Parameters

struct ib_cq * cq The CQ to return.

unsigned int nr_cqe The max number of cqes that the user had requested.

int **ib_cm_listen**(struct ib_cm_id * cm_id, __be64 service_id, __be64 service_mask)
Initiates listening on the specified service ID for connection and service ID resolution requests.

Parameters

struct ib_cm_id * cm_id Connection identifier associated with the listen request.

__be64 service_id Service identifier matched against incoming connection and service ID resolution requests. The service ID should be specified network-

byte order. If set to IB_CM_ASSIGN_SERVICE_ID, the CM will assign a service ID to the caller.

__be64 service_mask Mask applied to service ID used to listen across a range of service IDs. If set to 0, the service ID is matched exactly. This parameter is ignored if `service_id` is set to IB_CM_ASSIGN_SERVICE_ID.

struct ib_cm_id * **ib_cm_insert_listen**(struct ib_device * device, ib_cm_handler cm_handler, __be64 service_id)

Parameters

struct ib_device * device Device associated with the cm_id. All related communication will be associated with the specified device.

ib_cm_handler cm_handler Callback invoked to notify the user of CM events.

__be64 service_id Service identifier matched against incoming connection and service ID resolution requests. The service ID should be specified network-byte order. If set to IB_CM_ASSIGN_SERVICE_ID, the CM will assign a service ID to the caller.

Description

If there's an existing ID listening on that same device and service ID, return it.

Callers should call ib_destroy_cm_id when done with the listener ID.

int **rdma_rw_ctx_init**(struct rdma_rw_ctx * ctx, struct ib_qp * qp, u8 port_num, struct scatterlist * sg, u32 sg_cnt, u32 sg_offset, u64 remote_addr, u32 rkey, enum dma_data_direction dir)

initialize a RDMA READ/WRITE context

Parameters

struct rdma_rw_ctx * ctx context to initialize

struct ib_qp * qp queue pair to operate on

u8 port_num port num to which the connection is bound

struct scatterlist * sg scatterlist to READ/WRITE from/to

u32 sg_cnt number of entries in **sg**

u32 sg_offset current byte offset into **sg**

u64 remote_addr remote address to read/write (relative to **rkey**)

u32 rkey remote key to operate on

enum dma_data_direction dir `DMA_TO_DEVICE` for RDMA WRITE, `DMA_FROM_DEVICE` for RDMA READ

Description

Returns the number of WQEs that will be needed on the workqueue if successful, or a negative error code.

int **rdma_rw_ctx_signature_init**(struct rdma_rw_ctx * ctx, struct ib_qp * qp, u8 port_num, struct scatterlist * sg, u32 sg_cnt, struct scatterlist * prot_sg, u32 prot_sg_cnt, struct ib_sig_attrs * sig_attrs, u64 remote_addr, u32 rkey, enum dma_data_direction dir)

initialize a RW context with signature offload

Parameters

struct rdma_rw_ctx * ctx context to initialize

struct ib_qp * qp queue pair to operate on

u8 port_num port num to which the connection is bound

struct scatterlist * sg scatterlist to READ/WRITE from/to

u32 sg_cnt number of entries in **sg**

struct scatterlist * prot_sg scatterlist to READ/WRITE protection information from/to

u32 prot_sg_cnt number of entries in **prot_sg**

struct ib_sig_attrs * sig_attrs signature offloading algorithms

u64 remote_addr remote address to read/write (relative to **rkey**)

u32 rkey remote key to operate on

enum dma_data_direction dir `DMA_TO_DEVICE` for RDMA WRITE, `DMA_FROM_DEVICE` for RDMA READ

Description

Returns the number of WQEs that will be needed on the workqueue if successful, or a negative error code.

struct ib_send_wr * **rdma_rw_ctx_wrs**(struct rdma_rw_ctx * ctx, struct ib_qp * qp, u8 port_num, struct ib_cqe * cqe, struct ib_send_wr * chain_wr)

return chain of WRs for a RDMA READ or WRITE operation

Parameters

struct rdma_rw_ctx * ctx context to operate on

struct ib_qp * qp queue pair to operate on

u8 port_num port num to which the connection is bound

struct ib_cqe * cqe completion queue entry for the last WR

struct ib_send_wr * chain_wr WR to append to the posted chain

Description

Return the WR chain for the set of RDMA READ/WRITE operations described by **ctx**, as well as any memory registration operations needed. If **chain_wr** is non-NULL the WR it points to will be appended to the chain of WRs posted. If **chain_wr** is not set **cqe** must be set so that the caller gets a completion notification.

int **rdma_rw_ctx_post**(struct rdma_rw_ctx * ctx, struct ib_qp * qp, u8 port_num, struct ib_cqe * cqe, struct ib_send_wr * chain_wr)

post a RDMA READ or RDMA WRITE operation

Parameters

struct rdma_rw_ctx * ctx context to operate on

struct ib_qp * qp queue pair to operate on

u8 port_num port num to which the connection is bound

struct ib_cqe * cqe completion queue entry for the last WR

struct ib_send_wr * chain_wr WR to append to the posted chain

Description

Post the set of RDMA READ/WRITE operations described by **ctx**, as well as any memory registration operations needed. If **chain_wr** is non-NULL the WR it points to will be appended to the chain of WRs posted. If **chain_wr** is not set **cqe** must be set so that the caller gets a completion notification.

void **rdma_rw_ctx_destroy**(struct rdma_rw_ctx * ctx, struct ib_qp * qp, u8 port_num, struct scatterlist * sg, u32 sg_cnt, enum dma_data_direction dir)

release all resources allocated by rdma_rw_ctx_init

Parameters

struct rdma_rw_ctx * ctx context to release

struct ib_qp * qp queue pair to operate on

u8 port_num port num to which the connection is bound

struct scatterlist * sg scatterlist that was used for the READ/WRITE

u32 sg_cnt number of entries in **sg**

enum dma_data_direction dir `DMA_TO_DEVICE` for RDMA WRITE, `DMA_FROM_DEVICE` for RDMA READ

void **rdma_rw_ctx_destroy_signature**(struct rdma_rw_ctx * ctx, struct ib_qp * qp, u8 port_num, struct scatterlist * sg, u32 sg_cnt, struct scatterlist * prot_sg, u32 prot_sg_cnt, enum dma_data_direction dir)

release all resources allocated by rdma_rw_ctx_signature_init

Parameters

struct rdma_rw_ctx * ctx context to release

struct ib_qp * qp queue pair to operate on

u8 port_num port num to which the connection is bound

struct scatterlist * sg scatterlist that was used for the READ/WRITE

u32 sg_cnt number of entries in **sg**

struct scatterlist * prot_sg scatterlist that was used for the READ/WRITE of the PI

u32 prot_sg_cnt number of entries in **prot_sg**

enum dma_data_direction dir DMA_TO_DEVICE for RDMA WRITE, DMA_FROM_DEVICE for RDMA READ

unsigned int **rdma_rw_mr_factor**(struct ib_device * device, u8 port_num, unsigned int maxpages)
return number of MRs required for a payload

Parameters

struct ib_device * device device handling the connection

u8 port_num port num to which the connection is bound

unsigned int maxpages maximum payload pages per rdma_rw_ctx

Description

Returns the number of MRs the device requires to move **maxpayload** bytes. The returned value is used during transport creation to compute max_rdma_ctxts and the size of the transport's Send and Send Completion Queues.

bool **rdma_dev_access_netns**(const struct ib_device * dev, const struct net * net)
Return whether an rdma device can be accessed from a specified net namespace or not.

Parameters

const struct ib_device * dev Pointer to rdma device which needs to be checked

const struct net * net Pointer to net namesapce for which access to be checked

Description

When the rdma device is in shared mode, it ignores the net namespace. When the rdma device is exclusive to a net namespace, rdma device net namespace is checked against the specified one.

void **ib_device_put**(struct ib_device * device)
Release IB device reference

Parameters

struct ib_device * device device whose reference to be released

Description

ib_device_put() releases reference to the IB device to allow it to be unregistered and eventually free.

struct ib_device * **ib_device_get_by_name**(const char * name, enum rdma_driver_id driver_id)
Find an IB device by name

Parameters

const char * name The name to look for

enum rdma_driver_id driver_id The driver ID that must match (RDMA_DRIVER_UNKNOWN matches all)

Description

Find and hold an ib_device by its name. The caller must call `ib_device_put()` on the returned pointer.

struct ib_device * **_ib_alloc_device**(size_t size)
allocate an IB device struct

Parameters

size_t size size of structure to allocate

Description

Low-level drivers should use ib_alloc_device() to allocate `struct ib_device`. **size** is the size of the structure to be allocated, including any private data used by the low-level driver. `ib_dealloc_device()` must be used to free structures allocated with ib_alloc_device().

void **ib_dealloc_device**(struct ib_device * device)
free an IB device struct

Parameters

struct ib_device * device structure to free

Description

Free a structure allocated with ib_alloc_device().

int **ib_register_device**(struct ib_device * device, const char * name)
Register an IB device with IB core

Parameters

struct ib_device * device Device to register

const char * name unique string device name. This may include a '%' which will cause a unique index to be added to the passed device name.

Description

Low-level drivers use `ib_register_device()` to register their devices with the IB core. All registered clients will receive a callback for each device that is added. **device** must be allocated with ib_alloc_device().

If the driver uses ops.dealloc_driver and calls any `ib_unregister_device()` asynchronously then the device pointer may become freed as soon as this function returns.

void **ib_unregister_device**(struct ib_device * ib_dev)
Unregister an IB device

Parameters

struct ib_device * ib_dev The device to unregister

Description

Unregister an IB device. All clients will receive a remove callback.

Callers should call this routine only once, and protect against races with registration. Typically it should only be called as part of a remove callback in an implementation of driver core' s struct device_driver and related.

If ops.dealloc_driver is used then ib_dev will be freed upon return from this function.

void **ib_unregister_device_and_put**(struct ib_device * ib_dev)
Unregister a device while holding a 'get'

Parameters

struct ib_device * ib_dev The device to unregister

Description

This is the same as `ib_unregister_device()`, except it includes an internal `ib_device_put()` that should match a 'get' obtained by the caller.

It is safe to call this routine concurrently from multiple threads while holding the 'get' . When the function returns the device is fully unregistered.

Drivers using this flow MUST use the driver_unregister callback to clean up their resources associated with the device and dealloc it.

void **ib_unregister_driver**(enum rdma_driver_id driver_id)
Unregister all IB devices for a driver

Parameters

enum rdma_driver_id driver_id The driver to unregister

Description

This implements a fence for device unregistration. It only returns once all devices associated with the driver_id have fully completed their unregistration and returned from ib_unregister_device*().

If device' s are not yet unregistered it goes ahead and starts unregistering them.

This does not block creation of new devices with the given driver_id, that is the responsibility of the caller.

void **ib_unregister_device_queued**(struct ib_device * ib_dev)
Unregister a device using a work queue

Parameters

struct ib_device * ib_dev The device to unregister

Description

This schedules an asynchronous unregistration using a WQ for the device. A driver should use this to avoid holding locks while doing unregistration, such as holding the RTNL lock.

Drivers using this API must use ib_unregister_driver before module unload to ensure that all scheduled unregistrations have completed.

int **ib_register_client**(struct ib_client * client)
Register an IB client

Parameters

struct ib_client * client Client to register

Description

Upper level users of the IB drivers can use `ib_register_client()` to register callbacks for IB device addition and removal. When an IB device is added, each registered client's add method will be called (in the order the clients were registered), and when a device is removed, each client's remove method will be called (in the reverse order that clients were registered). In addition, when `ib_register_client()` is called, the client will receive an add callback for all devices already registered.

void **ib_unregister_client**(struct ib_client * client)
Unregister an IB client

Parameters

struct ib_client * client Client to unregister

Description

Upper level users use `ib_unregister_client()` to remove their client registration. When `ib_unregister_client()` is called, the client will receive a remove callback for each IB device still registered.

This is a full fence, once it returns no client callbacks will be called, or are running in another thread.

void **ib_set_client_data**(struct ib_device * device, struct ib_client * client,
void * data)
Set IB client context

Parameters

struct ib_device * device Device to set context for

struct ib_client * client Client to set context for

void * data Context to set

Description

`ib_set_client_data()` sets client context data that can be retrieved with ib_get_client_data(). This can only be called while the client is registered to the device, once the ib_client remove() callback returns this cannot be called.

void **ib_register_event_handler**(struct ib_event_handler * event_handler)
Register an IB event handler

Parameters

struct ib_event_handler * event_handler Handler to register

Description

`ib_register_event_handler()` registers an event handler that will be called back when asynchronous IB events occur (as defined in chapter 11 of the InfiniBand Architecture Specification). This callback occurs in workqueue context.

void **ib_unregister_event_handler**(struct ib_event_handler * event_handler)

Unregister an event handler

Parameters

struct ib_event_handler * event_handler Handler to unregister

Description

Unregister an event handler registered with `ib_register_event_handler()`.

int **ib_query_port**(struct ib_device * device, u8 port_num, struct ib_port_attr * port_attr)

Query IB port attributes

Parameters

struct ib_device * device Device to query

u8 port_num Port number to query

struct ib_port_attr * port_attr Port attributes

Description

`ib_query_port()` returns the attributes of a port through the **port_attr** pointer.

int **ib_device_set_netdev**(struct ib_device * ib_dev, struct net_device * ndev, unsigned int port)

Associate the ib_dev with an underlying net_device

Parameters

struct ib_device * ib_dev Device to modify

struct net_device * ndev net_device to affiliate, may be NULL

unsigned int port IB port the net_device is connected to

Description

Drivers should use this to link the ib_device to a netdev so the netdev shows up in interfaces like ib_enum_roce_netdev. Only one netdev may be affiliated with any port.

The caller must ensure that the given ndev is not unregistered or unregistering, and that either the ib_device is unregistered or `ib_device_set_netdev()` is called with NULL when the ndev sends a NETDEV_UNREGISTER event.

struct ib_device * **ib_device_get_by_netdev**(struct net_device * ndev, enum rdma_driver_id driver_id)

Find an IB device associated with a netdev

Parameters

struct net_device * ndev netdev to locate

enum rdma_driver_id driver_id The driver ID that must match (RDMA_DRIVER_UNKNOWN matches all)

Description

Find and hold an ib_device that is associated with a netdev via `ib_device_set_netdev()`. The caller must call `ib_device_put()` on the returned pointer.

int **ib_query_pkey**(struct ib_device * device, u8 port_num, u16 index, u16 * pkey)
Get P_Key table entry

Parameters

struct ib_device * device Device to query

u8 port_num Port number to query

u16 index P_Key table index to query

u16 * pkey Returned P_Key

Description

`ib_query_pkey()` fetches the specified P_Key table entry.

int **ib_modify_device**(struct ib_device * device, int device_modify_mask, struct ib_device_modify * device_modify)
Change IB device attributes

Parameters

struct ib_device * device Device to modify

int device_modify_mask Mask of attributes to change

struct ib_device_modify * device_modify New attribute values

Description

`ib_modify_device()` changes a device's attributes as specified by the **device_modify_mask** and **device_modify** structure.

int **ib_modify_port**(struct ib_device * device, u8 port_num, int port_modify_mask, struct ib_port_modify * port_modify)
Modifies the attributes for the specified port.

Parameters

struct ib_device * device The device to modify.

u8 port_num The number of the port to modify.

int port_modify_mask Mask used to specify which attributes of the port to change.

struct ib_port_modify * port_modify New attribute values for the port.

Description

`ib_modify_port()` changes a port's attributes as specified by the **port_modify_mask** and **port_modify** structure.

int **ib_find_gid**(struct ib_device * device, union ib_gid * gid, u8 * port_num, u16 * index)

Returns the port number and GID table index where a specified GID value occurs. Its searches only for IB link layer.

Parameters

struct ib_device * device The device to query.

union ib_gid * gid The GID value to search for.

u8 * port_num The port number of the device where the GID value was found.

u16 * index The index into the GID table where the GID was found. This parameter may be NULL.

int **ib_find_pkey**(struct ib_device * device, u8 port_num, u16 pkey, u16 * index)

Returns the PKey table index where a specified PKey value occurs.

Parameters

struct ib_device * device The device to query.

u8 port_num The port number of the device to search for the PKey.

u16 pkey The PKey value to search for.

u16 * index The index into the PKey table where the PKey was found.

struct net_device * **ib_get_net_dev_by_params**(struct ib_device * dev, u8 port, u16 pkey, const union ib_gid * gid, const struct sockaddr * addr)

Return the appropriate net_dev for a received CM request

Parameters

struct ib_device * dev An RDMA device on which the request has been received.

u8 port Port number on the RDMA device.

u16 pkey The Pkey the request came on.

const union ib_gid * gid A GID that the net_dev uses to communicate.

const struct sockaddr * addr Contains the IP address that the request specified as its destination.

struct ib_pd * **__ib_alloc_pd**(struct ib_device * device, unsigned int flags, const char * caller)

Allocates an unused protection domain.

Parameters

struct ib_device * device The device on which to allocate the protection domain.

unsigned int flags protection domain flags

const char * caller caller's build-time module name

Description

A protection domain object provides an association between QPs, shared receive queues, address handles, memory regions, and memory windows.

Every PD has a local_dma_lkey which can be used as the lkey value for local memory operations.

void **ib_dealloc_pd_user**(struct ib_pd * pd, struct ib_udata * udata)
Deallocates a protection domain.

Parameters

struct ib_pd * pd The protection domain to deallocate.

struct ib_udata * udata Valid user data or NULL for kernel object

Description

It is an error to call this function while any resources in the pd still exist. The caller is responsible to synchronously destroy them and guarantee no new allocations will happen.

void **rdma_copy_ah_attr**(struct rdma_ah_attr * dest, const struct rdma_ah_attr * src)
Copy rdma ah attribute from source to destination.

Parameters

struct rdma_ah_attr * dest Pointer to destination ah_attr. Contents of the destination pointer is assumed to be invalid and attribute are overwritten.

const struct rdma_ah_attr * src Pointer to source ah_attr.

void **rdma_replace_ah_attr**(struct rdma_ah_attr * old, const struct rdma_ah_attr * new)
Replace valid ah_attr with new new one.

Parameters

struct rdma_ah_attr * old Pointer to existing ah_attr which needs to be replaced. old is assumed to be valid or zero'd

const struct rdma_ah_attr * new Pointer to the new ah_attr.

Description

`rdma_replace_ah_attr()` first releases any reference in the old ah_attr if old the ah_attr is valid; after that it copies the new attribute and holds the reference to the replaced ah_attr.

void **rdma_move_ah_attr**(struct rdma_ah_attr * dest, struct rdma_ah_attr * src)
Move ah_attr pointed by source to destination.

Parameters

struct rdma_ah_attr * dest Pointer to destination ah_attr to copy to. dest is assumed to be valid or zero'd

struct rdma_ah_attr * src Pointer to the new ah_attr.

Description

`rdma_move_ah_attr()` first releases any reference in the destination ah_attr if it is valid. This also transfers ownership of internal references from src to dest, making src invalid in the process. No new reference of the src ah_attr is taken.

struct ib_ah * **rdma_create_ah**(struct ib_pd * pd, struct rdma_ah_attr * ah_attr, u32 flags)

Creates an address handle for the given address vector.

Parameters

struct ib_pd * pd The protection domain associated with the address handle.

struct rdma_ah_attr * ah_attr The attributes of the address vector.

u32 flags Create address handle flags (see enum rdma_create_ah_flags).

Description

It returns 0 on success and returns appropriate error code on error. The address handle is used to reference a local or global destination in all UD QP post sends.

struct ib_ah * **rdma_create_user_ah**(struct ib_pd * pd, struct rdma_ah_attr * ah_attr, struct ib_udata * udata)

Creates an address handle for the given address vector. It resolves destination mac address for ah attribute of RoCE type.

Parameters

struct ib_pd * pd The protection domain associated with the address handle.

struct rdma_ah_attr * ah_attr The attributes of the address vector.

struct ib_udata * udata pointer to user's input output buffer information need by provider driver.

Description

It returns 0 on success and returns appropriate error code on error. The address handle is used to reference a local or global destination in all UD QP post sends.

void **rdma_move_grh_sgid_attr**(struct rdma_ah_attr * attr, union ib_gid * dgid, u32 flow_label, u8 hop_limit, u8 traffic_class, const struct ib_gid_attr * sgid_attr)

Sets the sgid attribute of GRH, taking ownership of the reference

Parameters

struct rdma_ah_attr * attr Pointer to AH attribute structure

union ib_gid * dgid Destination GID

u32 flow_label Flow label

u8 hop_limit Hop limit

u8 traffic_class traffic class

const struct ib_gid_attr * sgid_attr Pointer to SGID attribute

Description

This takes ownership of the sgid_attr reference. The caller must ensure `rdma_destroy_ah_attr()` is called before destroying the rdma_ah_attr after calling this function.

void **rdma_destroy_ah_attr**(struct rdma_ah_attr * ah_attr)
Release reference to SGID attribute of ah attribute.

Parameters

struct rdma_ah_attr * ah_attr Pointer to ah attribute

Description

Release reference to the SGID attribute of the ah attribute if it is non NULL. It is safe to call this multiple times, and safe to call it on a zero initialized ah_attr.

struct ib_srq * **ib_create_srq_user**(struct ib_pd * pd, struct ib_srq_init_attr * srq_init_attr, struct ib_usrq_object * uobject, struct ib_udata * udata)
Creates a SRQ associated with the specified protection domain.

Parameters

struct ib_pd * pd The protection domain associated with the SRQ.

struct ib_srq_init_attr * srq_init_attr A list of initial attributes required to create the SRQ. If SRQ creation succeeds, then the attributes are updated to the actual capabilities of the created SRQ. **uobject** - uobject pointer if this is not a kernel SRQ **udata** - udata pointer if this is not a kernel SRQ

struct ib_usrq_object * uobject undescribed

struct ib_udata * udata undescribed

Description

srq_attr->max_wr and srq_attr->max_sge are read the determine the requested size of the SRQ, and set to the actual values allocated on return. If ib_create_srq() succeeds, then max_wr and max_sge will always be at least as large as the requested values.

struct ib_qp * **ib_create_qp**(struct ib_pd * pd, struct ib_qp_init_attr * qp_init_attr)
Creates a kernel QP associated with the specified protection domain.

Parameters

struct ib_pd * pd The protection domain associated with the QP.

struct ib_qp_init_attr * qp_init_attr A list of initial attributes required to create the QP. If QP creation succeeds, then the attributes are updated to the actual capabilities of the created QP.

NOTE

for user qp use ib_create_qp_user with valid udata!

int **ib_modify_qp_with_udata**(struct ib_qp * ib_qp, struct ib_qp_attr * attr, int attr_mask, struct ib_udata * udata)
Modifies the attributes for the specified QP.

Parameters

struct ib_qp * ib_qp The QP to modify.

struct ib_qp_attr * attr On input, specifies the QP attributes to modify. On output, the current values of selected QP attributes are returned.

int attr_mask A bit-mask used to specify which attributes of the QP are being modified.

struct ib_udata * udata pointer to user's input output buffer information are being modified. It returns 0 on success and returns appropriate error code on error.

struct ib_mr * **ib_alloc_mr_user**(struct ib_pd * pd, enum ib_mr_type mr_type, u32 max_num_sg, struct ib_udata * udata)

Allocates a memory region

Parameters

struct ib_pd * pd protection domain associated with the region

enum ib_mr_type mr_type memory region type

u32 max_num_sg maximum sg entries available for registration.

struct ib_udata * udata user data or null for kernel objects

Notes

Memory registeration page/sg lists must not exceed max_num_sg. For mr_type IB_MR_TYPE_MEM_REG, the total length cannot exceed max_num_sg * used_page_size.

struct ib_mr * **ib_alloc_mr_integrity**(struct ib_pd * pd, u32 max_num_data_sg, u32 max_num_meta_sg)

Allocates an integrity memory region

Parameters

struct ib_pd * pd protection domain associated with the region

u32 max_num_data_sg maximum data sg entries available for registration

u32 max_num_meta_sg maximum metadata sg entries available for registration

Notes

Memory registration page/sg lists must not exceed max_num_sg, also the integrity page/sg lists must not exceed max_num_meta_sg.

struct ib_wq * **ib_create_wq**(struct ib_pd * pd, struct ib_wq_init_attr * wq_attr)

Creates a WQ associated with the specified protection domain.

Parameters

struct ib_pd * pd The protection domain associated with the WQ.

struct ib_wq_init_attr * wq_attr A list of initial attributes required to create the WQ. If WQ creation succeeds, then the attributes are updated to the actual capabilities of the created WQ.

Description

wq_attr->max_wr and wq_attr->max_sge determine the requested size of the WQ, and set to the actual values allocated on return. If `ib_create_wq()` succeeds, then max_wr and max_sge will always be at least as large as the requested values.

int **ib_destroy_wq**(struct ib_wq * wq, struct ib_udata * udata)
Destroys the specified user WQ.

Parameters

struct ib_wq * wq The WQ to destroy.

struct ib_udata * udata Valid user data

int **ib_modify_wq**(struct ib_wq * wq, struct ib_wq_attr * wq_attr, u32 wq_attr_mask)
Modifies the specified WQ.

Parameters

struct ib_wq * wq The WQ to modify.

struct ib_wq_attr * wq_attr On input, specifies the WQ attributes to modify.

u32 wq_attr_mask A bit-mask used to specify which attributes of the WQ are being modified. On output, the current values of selected WQ attributes are returned.

int **ib_map_mr_sg_pi**(struct ib_mr * mr, struct scatterlist * data_sg, int data_sg_nents, unsigned int * data_sg_offset, struct scatterlist * meta_sg, int meta_sg_nents, unsigned int * meta_sg_offset, unsigned int page_size)
Map the dma mapped SG lists for PI (protection information) and set an appropriate memory region for registration.

Parameters

struct ib_mr * mr memory region

struct scatterlist * data_sg dma mapped scatterlist for data

int data_sg_nents number of entries in data_sg

unsigned int * data_sg_offset offset in bytes into data_sg

struct scatterlist * meta_sg dma mapped scatterlist for metadata

int meta_sg_nents number of entries in meta_sg

unsigned int * meta_sg_offset offset in bytes into meta_sg

unsigned int page_size page vector desired page size

Description

Constraints: - The MR must be allocated with type IB_MR_TYPE_INTEGRITY.

After this completes successfully, the memory region is ready for registration.

Return

0 on success.

int **ib_map_mr_sg**(struct ib_mr * mr, struct scatterlist * sg, int sg_nents, unsigned int * sg_offset, unsigned int page_size)
Map the largest prefix of a dma mapped SG list and set it the memory region.

Parameters

struct ib_mr * mr memory region

struct scatterlist * sg dma mapped scatterlist

int sg_nents number of entries in sg

unsigned int * sg_offset offset in bytes into sg

unsigned int page_size page vector desired page size

Description

Constraints:

- The first sg element is allowed to have an offset.
- Each sg element must either be aligned to page_size or virtually contiguous to the previous element. In case an sg element has a non-contiguous offset, the mapping prefix will not include it.
- The last sg element is allowed to have length less than page_size.
- If sg_nents total byte length exceeds the mr max_num_sge * page_size then only max_num_sg entries will be mapped.
- If the MR was allocated with type IB_MR_TYPE_SG_GAPS, none of these constraints holds and the page_size argument is ignored.

Returns the number of sg elements that were mapped to the memory region.

After this completes successfully, the memory region is ready for registration.

int **ib_sg_to_pages**(struct ib_mr * mr, struct scatterlist * sgl, int sg_nents, unsigned int * sg_offset_p, int (*set_page)(struct ib_mr *, u64))
Convert the largest prefix of a sg list to a page vector

Parameters

struct ib_mr * mr memory region

struct scatterlist * sgl dma mapped scatterlist

int sg_nents number of entries in sg

unsigned int * sg_offset_p

IN	start offset in bytes into sg
OUT	offset in bytes for element n of the sg of the first byte that has not been processed where n is the return value of this function.

int (*)(struct ib_mr *, u64) set_page driver page assignment function pointer

Description

Core service helper for drivers to convert the largest prefix of given sg list to a page vector. The sg list prefix converted is the prefix that meet the requirements of ib_map_mr_sg.

Returns the number of sg elements that were assigned to a page vector.

void **ib_drain_sq**(struct ib_qp * qp)
Block until all SQ CQEs have been consumed by the application.

Parameters

struct ib_qp * qp queue pair to drain

Description

If the device has a provider-specific drain function, then call that. Otherwise call the generic drain function __ib_drain_sq().

The caller must:

ensure there is room in the CQ and SQ for the drain work request and completion.

allocate the CQ using ib_alloc_cq().

ensure that there are no other contexts that are posting WRs concurrently. Otherwise the drain is not guaranteed.

void **ib_drain_rq**(struct ib_qp * qp)
Block until all RQ CQEs have been consumed by the application.

Parameters

struct ib_qp * qp queue pair to drain

Description

If the device has a provider-specific drain function, then call that. Otherwise call the generic drain function __ib_drain_rq().

The caller must:

ensure there is room in the CQ and RQ for the drain work request and completion.

allocate the CQ using ib_alloc_cq().

ensure that there are no other contexts that are posting WRs concurrently. Otherwise the drain is not guaranteed.

void **ib_drain_qp**(struct ib_qp * qp)
Block until all CQEs have been consumed by the application on both the RQ and SQ.

Parameters

struct ib_qp * qp queue pair to drain

Description

The caller must:

ensure there is room in the CQ(s), SQ, and RQ for drain work requests and completions.

allocate the CQs using ib_alloc_cq().

ensure that there are no other contexts that are posting WRs concurrently. Otherwise the drain is not guaranteed.

void **ib_pack**(const struct ib_field * desc, int desc_len, void * structure, void * buf)
Pack a structure into a buffer

Parameters

const struct ib_field * desc Array of structure field descriptions

int desc_len Number of entries in **desc**

void * structure Structure to pack from

void * buf Buffer to pack into

Description

ib_pack() packs a list of structure fields into a buffer, controlled by the array of fields in **desc**.

void **ib_unpack**(const struct ib_field * desc, int desc_len, void * buf, void * structure)
Unpack a buffer into a structure

Parameters

const struct ib_field * desc Array of structure field descriptions

int desc_len Number of entries in **desc**

void * buf Buffer to unpack from

void * structure Structure to unpack into

Description

ib_pack() unpacks a list of structure fields from a buffer, controlled by the array of fields in **desc**.

void **ib_sa_cancel_query**(int id, struct ib_sa_query * query)
try to cancel an SA query

Parameters

int id ID of query to cancel

struct ib_sa_query * query query pointer to cancel

Description

Try to cancel an SA query. If the id and query don't match up or the query has already completed, nothing is done. Otherwise the query is canceled and will complete with a status of -EINTR.

int **ib_init_ah_attr_from_path**(struct ib_device * device, u8 port_num, struct sa_path_rec * rec, struct rdma_ah_attr * ah_attr, const struct ib_gid_attr * gid_attr)
Initialize address handle attributes based on an SA path record.

Parameters

struct ib_device * device Device associated ah attributes initialization.

u8 port_num Port on the specified device.

struct sa_path_rec * rec path record entry to use for ah attributes initialization.

struct rdma_ah_attr * ah_attr address handle attributes to initialization from path record.

const struct ib_gid_attr * gid_attr SGID attribute to consider during initialization.

Description

When `ib_init_ah_attr_from_path()` returns success, (a) for IB link layer it optionally contains a reference to SGID attribute when GRH is present for IB link layer. (b) for RoCE link layer it contains a reference to SGID attribute. User must invoke `rdma_destroy_ah_attr()` to release reference to SGID attributes which are initialized using `ib_init_ah_attr_from_path()`.

int **ib_sa_path_rec_get**(struct ib_sa_client * client, struct ib_device * device, u8 port_num, struct sa_path_rec * rec, ib_sa_comp_mask comp_mask, unsigned long timeout_ms, gfp_t gfp_mask, void (*callback)(int status, struct sa_path_rec *resp, void *context), void * context, struct ib_sa_query ** sa_query)

Start a Path get query

Parameters

struct ib_sa_client * client SA client

struct ib_device * device device to send query on

u8 port_num port number to send query on

struct sa_path_rec * rec Path Record to send in query

ib_sa_comp_mask comp_mask component mask to send in query

unsigned long timeout_ms time to wait for response

gfp_t gfp_mask GFP mask to use for internal allocations

void (*)(int status, struct sa_path_rec *resp, void *context) callback function called when query completes, times out or is canceled

void * context opaque user context passed to callback

struct ib_sa_query ** sa_query query context, used to cancel query

Description

Send a Path Record Get query to the SA to look up a path. The callback function will be called when the query completes (or fails); status is 0 for a successful response, -EINTR if the query is canceled, -ETIMEDOUT is the query timed out, or -EIO if an error occurred sending the query. The resp parameter of the callback is only valid if status is 0.

If the return value of `ib_sa_path_rec_get()` is negative, it is an error code. Otherwise it is a query ID that can be used to cancel the query.

int **ib_sa_service_rec_query**(struct ib_sa_client * client, struct ib_device * device, u8 port_num, u8 method, struct ib_sa_service_rec * rec, ib_sa_comp_mask comp_mask, unsigned long timeout_ms, gfp_t gfp_mask, void (*callback)(int status, struct ib_sa_service_rec *resp, void *context), void * context, struct ib_sa_query ** sa_query)

Start Service Record operation

Parameters

struct ib_sa_client * client SA client

struct ib_device * device device to send request on

u8 port_num port number to send request on

u8 method SA method - should be get, set, or delete

struct ib_sa_service_rec * rec Service Record to send in request

ib_sa_comp_mask comp_mask component mask to send in request

unsigned long timeout_ms time to wait for response

gfp_t gfp_mask GFP mask to use for internal allocations

void (*)(int status, struct ib_sa_service_rec *resp, void *context) callback function called when request completes, times out or is canceled

void * context opaque user context passed to callback

struct ib_sa_query ** sa_query request context, used to cancel request

Description

Send a Service Record set/get/delete to the SA to register, unregister or query a service record. The callback function will be called when the request completes (or fails); status is 0 for a successful response, -EINTR if the query is canceled, -ETIMEDOUT is the query timed out, or -EIO if an error occurred sending the query. The resp parameter of the callback is only valid if status is 0.

If the return value of `ib_sa_service_rec_query()` is negative, it is an error code. Otherwise it is a request ID that can be used to cancel the query.

int **ib_ud_header_init**(int payload_bytes, int lrh_present, int eth_present, int vlan_present, int grh_present, int ip_version, int udp_present, int immediate_present, struct ib_ud_header * header)

Initialize UD header structure

Parameters

int payload_bytes Length of packet payload

int lrh_present specify if LRH is present

int eth_present specify if Eth header is present

int vlan_present packet is tagged vlan

int grh_present GRH flag (if non-zero, GRH will be included)

int ip_version if non-zero, IP header, V4 or V6, will be included

int udp_present if non-zero, UDP header will be included

int immediate_present specify if immediate data is present

struct ib_ud_header * header Structure to initialize

int **ib_ud_header_pack**(struct ib_ud_header * header, void * buf)
Pack UD header struct into wire format

Parameters

struct ib_ud_header * header UD header struct

void * buf Buffer to pack into

Description

`ib_ud_header_pack()` packs the UD header structure **header** into wire format in the buffer **buf**.

int **ib_ud_header_unpack**(void * buf, struct ib_ud_header * header)
Unpack UD header struct from wire format

Parameters

void * buf Buffer to pack into

struct ib_ud_header * header UD header struct

Description

`ib_ud_header_pack()` unpacks the UD header structure **header** from wire format in the buffer **buf**.

unsigned long **ib_umem_find_best_pgsz**(struct ib_umem * umem, unsigned long pgsz_bitmap, unsigned long virt)
Find best HW page size to use for this MR

Parameters

struct ib_umem * umem umem struct

unsigned long pgsz_bitmap bitmap of HW supported page sizes

unsigned long virt IOVA

Description

This helper is intended for HW that support multiple page sizes but can do only a single page size in an MR.

Returns 0 if the umem requires page sizes not supported by the driver to be mapped. Drivers always supporting PAGE_SIZE or smaller will never see a 0 result.

struct ib_umem * **ib_umem_get**(struct ib_device * device, unsigned long addr, size_t size, int access)
Pin and DMA map userspace memory.

Parameters

struct ib_device * device IB device to connect UMEM

unsigned long addr userspace virtual address to start at

size_t size length of region to pin

int access IB_ACCESS_xxx flags for memory being pinned

void **ib_umem_release**(struct ib_umem * umem)
release memory pinned with ib_umem_get

Parameters

struct ib_umem * umem umem struct to release

struct ib_umem_odp * **ib_umem_odp_alloc_implicit**(struct ib_device * device, int access)
Allocate a parent implicit ODP umem

Parameters

struct ib_device * device IB device to create UMEM

int access ib_reg_mr access flags

Description

Implicit ODP umems do not have a VA range and do not have any page lists. They exist only to hold the per_mm reference to help the driver create children umems.

struct ib_umem_odp * **ib_umem_odp_alloc_child**(struct ib_umem_odp * root, unsigned long addr, size_t size, const struct mmu_interval_notifier_ops * ops)
Allocate a child ODP umem under an implicit parent ODP umem

Parameters

struct ib_umem_odp * root The parent umem enclosing the child. This must be allocated using ib_alloc_implicit_odp_umem()

unsigned long addr The starting userspace VA

size_t size The length of the userspace VA

const struct mmu_interval_notifier_ops * ops undescribed

struct ib_umem_odp * **ib_umem_odp_get**(struct ib_device * device, unsigned long addr, size_t size, int access, const struct mmu_interval_notifier_ops * ops)
Create a umem_odp for a userspace va

Parameters

struct ib_device * device IB device struct to get UMEM

unsigned long addr userspace virtual address to start at

size_t size length of region to pin

int access IB_ACCESS_xxx flags for memory being pinned

const struct mmu_interval_notifier_ops * ops undescribed

Description

The driver should use when the access flags indicate ODP memory. It avoids pinning, instead, stores the mm for future page fault handling in conjunction with MMU notifiers.

int **ib_umem_odp_map_dma_pages**(struct ib_umem_odp * umem_odp, u64 user_virt, u64 bcnt, u64 access_mask, unsigned long current_seq)

Pin and DMA map userspace memory in an ODP MR.

Parameters

struct ib_umem_odp * umem_odp the umem to map and pin

u64 user_virt the address from which we need to map.

u64 bcnt the minimal number of bytes to pin and map. The mapping might be bigger due to alignment, and may also be smaller in case of an error pinning or mapping a page. The actual pages mapped is returned in the return value.

u64 access_mask bit mask of the requested access permissions for the given range.

unsigned long current_seq the MMU notifiers sequance value for synchronization with invalidations. the sequance number is read from umem_odp->notifiers_seq before calling this function

Description

Pins the range of pages passed in the argument, and maps them to DMA addresses. The DMA addresses of the mapped pages is updated in umem_odp->dma_list.

Returns the number of pages mapped in success, negative error code for failure. An -EAGAIN error code is returned when a concurrent mmu notifier prevents the function from completing its task. An -ENOENT error code indicates that userspace process is being terminated and mm was already destroyed.

14.3 RDMA Verbs transport library

int **rvt_fast_reg_mr**(struct rvt_qp * qp, struct ib_mr * ibmr, u32 key, int access)

fast register physical MR

Parameters

struct rvt_qp * qp the queue pair where the work request comes from

struct ib_mr * ibmr the memory region to be registered

u32 key updated key for this memory region

int access access flags for this memory region

Description

Returns 0 on success.

int **rvt_invalidate_rkey**(struct rvt_qp * qp, u32 rkey)
invalidate an MR rkey

Parameters

struct rvt_qp * qp queue pair associated with the invalidate op

u32 rkey rkey to invalidate

Description

Returns 0 on success.

int **rvt_lkey_ok**(struct rvt_lkey_table * rkt, struct rvt_pd * pd, struct rvt_sge * isge, struct rvt_sge * last_sge, struct ib_sge * sge, int acc)
check IB SGE for validity and initialize

Parameters

struct rvt_lkey_table * rkt table containing lkey to check SGE against

struct rvt_pd * pd protection domain

struct rvt_sge * isge outgoing internal SGE

struct rvt_sge * last_sge last outgoing SGE written

struct ib_sge * sge SGE to check

int acc access flags

Description

Check the IB SGE for validity and initialize our internal version of it.

Increments the reference count when a new sge is stored.

Return

0 if compressed, 1 if added , otherwise returns -errno.

int **rvt_rkey_ok**(struct rvt_qp * qp, struct rvt_sge * sge, u32 len, u64 vaddr, u32 rkey, int acc)
check the IB virtual address, length, and RKEY

Parameters

struct rvt_qp * qp qp for validation

struct rvt_sge * sge SGE state

u32 len length of data

u64 vaddr virtual address to place data

u32 rkey rkey to check

int acc access flags

Return

1 if successful, otherwise 0.

Description

increments the reference count upon success

__be32 **rvt_compute_aeth**(struct rvt_qp * qp)
compute the AETH (syndrome + MSN)

Parameters

struct rvt_qp * qp the queue pair to compute the AETH for

Description

Returns the AETH.

void **rvt_get_credit**(struct rvt_qp * qp, u32 aeth)
flush the send work queue of a QP

Parameters

struct rvt_qp * qp the qp who’s send work queue to flush

u32 aeth the Acknowledge Extended Transport Header

Description

The QP s_lock should be held.

u32 **rvt_restart_sge**(struct rvt_sge_state * ss, struct rvt_swqe * wqe,
u32 len)
rewind the sge state for a wqe

Parameters

struct rvt_sge_state * ss the sge state pointer

struct rvt_swqe * wqe the wqe to rewind

u32 len the data length from the start of the wqe in bytes

Description

Returns the remaining data length.

int **rvt_check_ah**(struct ib_device * ibdev, struct rdma_ah_attr * ah_attr)
validate the attributes of AH

Parameters

struct ib_device * ibdev the ib device

struct rdma_ah_attr * ah_attr the attributes of the AH

Description

If driver supports a more detailed check_ah function call back to it otherwise just check the basics.

Return

0 on success

struct rvt_dev_info * **rvt_alloc_device**(size_t size, int nports)
allocate rdi

Parameters

size_t size how big of a structure to allocate

int nports number of ports to allocate array slots for

Description

Use IB core device alloc to allocate space for the rdi which is assumed to be inside of the ib_device. Any extra space that drivers require should be included in size.

We also allocate a port array based on the number of ports.

Return

pointer to allocated rdi

void **rvt_dealloc_device**(struct rvt_dev_info * rdi)
deallocate rdi

Parameters

struct rvt_dev_info * rdi structure to free

Description

Free a structure allocated with `rvt_alloc_device()`

int **rvt_register_device**(struct rvt_dev_info * rdi)
register a driver

Parameters

struct rvt_dev_info * rdi main dev structure for all of rdmavt operations

Description

It is up to drivers to allocate the rdi and fill in the appropriate information.

Return

0 on success otherwise an errno.

void **rvt_unregister_device**(struct rvt_dev_info * rdi)
remove a driver

Parameters

struct rvt_dev_info * rdi rvt dev struct

int **rvt_init_port**(struct rvt_dev_info * rdi, struct rvt_ibport * port, int port_index, u16 * pkey_table)
init internal data for driver port

Parameters

struct rvt_dev_info * rdi rvt_dev_info struct

struct rvt_ibport * port rvt port

int port_index 0 based index of ports, different from IB core port num

u16 * pkey_table pkey_table for **port**

Description

Keep track of a list of ports. No need to have a detach port. They persist until the driver goes away.

Return

always 0

bool **rvt_cq_enter**(struct rvt_cq * cq, struct ib_wc * entry, bool solicited)
add a new entry to the completion queue

Parameters

struct rvt_cq * cq completion queue

struct ib_wc * entry work completion entry to add

bool solicited true if **entry** is solicited

Description

This may be called with qp->s_lock held.

Return

return true on success, else return false if cq is full.

int **rvt_error_qp**(struct rvt_qp * qp, enum ib_wc_status err)
put a QP into the error state

Parameters

struct rvt_qp * qp the QP to put into the error state

enum ib_wc_status err the receive completion error to signal if a RWQE is active

Description

Flushes both send and receive work queues.

Return

true if last WQE event should be generated. The QP r_lock and s_lock should be held and interrupts disabled. If we are already in error state, just return.

int **rvt_get_rwqe**(struct rvt_qp * qp, bool wr_id_only)
copy the next RWQE into the QP’ s RWQE

Parameters

struct rvt_qp * qp the QP

bool wr_id_only update qp->r_wr_id only, not qp->r_sge

Description

Return -1 if there is a local error, 0 if no RWQE is available, otherwise return 1.

Can be called from interrupt level.

void **rvt_comm_est**(struct rvt_qp * qp)
handle trap with QP established

Parameters

struct rvt_qp * qp the QP

void **rvt_add_rnr_timer**(struct rvt_qp * qp, u32 aeth)
add/start an rnr timer on the QP

Parameters

struct rvt_qp * qp the QP

u32 aeth aeth of RNR timeout, simulated aeth for loopback

void **rvt_stop_rc_timers**(struct rvt_qp * qp)
stop all timers

Parameters

struct rvt_qp * qp the QP stop any pending timers

void **rvt_del_timers_sync**(struct rvt_qp * qp)
wait for any timeout routines to exit

Parameters

struct rvt_qp * qp the QP

struct rvt_qp_iter * **rvt_qp_iter_init**(struct rvt_dev_info * rdi, u64 v, void (*cb)(struct rvt_qp *qp, u64 v))
initial for QP iteration

Parameters

struct rvt_dev_info * rdi rvt devinfo

u64 v u64 value

void (*)(struct rvt_qp *qp, u64 v) cb user-defined callback

Description

This returns an iterator suitable for iterating QPs in the system.

The **cb** is a user-defined callback and **v** is a 64-bit value passed to and relevant for processing in the **cb**. An example use case would be to alter QP processing based on criteria not part of the rvt_qp.

Use cases that require memory allocation to succeed must preallocate appropriately.

Return

a pointer to an rvt_qp_iter or NULL

int **rvt_qp_iter_next**(struct rvt_qp_iter * iter)
return the next QP in iter

Parameters

struct rvt_qp_iter * iter the iterator

Description

Fine grained QP iterator suitable for use with debugfs seq_file mechanisms.

Updates iter->qp with the current QP when the return value is 0.

Return

0 - iter->qp is valid 1 - no more QPs

void **rvt_qp_iter**(struct rvt_dev_info * rdi, u64 v, void (*cb)(struct rvt_qp *qp, u64 v))
iterate all QPs

Parameters

struct rvt_dev_info * rdi rvt devinfo

u64 v a 64-bit value

void (*)(struct rvt_qp *qp, u64 v) cb a callback

Description

This provides a way for iterating all QPs.

The **cb** is a user-defined callback and **v** is a 64-bit value passed to and relevant for processing in the cb. An example use case would be to alter QP processing based on criteria not part of the rvt_qp.

The code has an internal iterator to simplify non seq_file use cases.

void **rvt_copy_sge**(struct rvt_qp * qp, struct rvt_sge_state * ss, void * data, u32 length, bool release, bool copy_last)
copy data to SGE memory

Parameters

struct rvt_qp * qp associated QP

struct rvt_sge_state * ss the SGE state

void * data the data to copy

u32 length the length of the data

bool release boolean to release MR

bool copy_last do a separate copy of the last 8 bytes

void **rvt_ruc_loopback**(struct rvt_qp * sqp)
handle UC and RC loopback requests

Parameters

struct rvt_qp * sqp the sending QP

Description

This is called from rvt_do_send() to forward a WQE addressed to the same HFI Note that although we are single threaded due to the send engine, we still have to protect against post_send(). We don't have to worry about receive interrupts since this is a connected protocol and all packets will pass through here.

struct rvt_mcast * **rvt_mcast_find**(struct rvt_ibport * ibp, union ib_gid * mgid, u16 lid)
search the global table for the given multicast GID/LID

Parameters

struct rvt_ibport * ibp the IB port structure

union ib_gid * mgid the multicast GID to search for

u16 lid the multicast LID portion of the multicast address (host order)

NOTE

It is valid to have 1 MLID with multiple MGIDs. It is not valid to have 1 MGID with multiple MLIDs.

Description

The caller is responsible for decrementing the reference count if found.

Return

NULL if not found.

14.4 Upper Layer Protocols

14.4.1 iSCSI Extensions for RDMA (iSER)

struct **iser_data_buf**
iSER data buffer

Definition

```
struct iser_data_buf {
  struct scatterlist *sg;
  int size;
  unsigned long      data_len;
  int dma_nents;
};
```

Members

sg pointer to the sg list

size num entries of this sg

data_len total beffer byte len

dma_nents returned by dma_map_sg

struct **iser_mem_reg**
iSER memory registration info

Definition

```
struct iser_mem_reg {
  struct ib_sge    sge;
  u32 rkey;
  void *mem_h;
};
```

Members

sge memory region sg element

rkey memory region remote key

mem_h pointer to registration context (FMR/Fastreg)

struct **iser_tx_desc**
iSER TX descriptor

Definition

```
struct iser_tx_desc {
  struct iser_ctrl             iser_header;
  struct iscsi_hdr             iscsi_header;
  enum iser_desc_type        type;
  u64 dma_addr;
  struct ib_sge                tx_sg[2];
  int num_sge;
  struct ib_cqe                cqe;
  bool mapped;
  struct ib_reg_wr             reg_wr;
  struct ib_send_wr            send_wr;
  struct ib_send_wr            inv_wr;
};
```

Members

iser_header iser header

iscsi_header iscsi header

type command/control/dataout

dma_addr header buffer dma_address

tx_sg sg[0] points to iser/iscsi headers sg[1] optionally points to either of immediate data unsolicited data-out or control

num_sge number sges used on this TX task

cqe completion handler

mapped Is the task header mapped

reg_wr registration WR

send_wr send WR

inv_wr invalidate WR

struct **iser_rx_desc**
iSER RX descriptor

Definition

```
struct iser_rx_desc {
  struct iser_ctrl             iser_header;
  struct iscsi_hdr             iscsi_header;
  char data[ISER_RECV_DATA_SEG_LEN];
  u64 dma_addr;
  struct ib_sge                rx_sg;
  struct ib_cqe                cqe;
  char pad[ISER_RX_PAD_SIZE];
};
```

Members

iser_header iser header

iscsi_header iscsi header

data received data segment

dma_addr receive buffer dma address

rx_sg ib_sge of receive buffer

cqe completion handler

pad for sense data TODO: Modify to maximum sense length supported

struct **iser_login_desc**
iSER login descriptor

Definition

```
struct iser_login_desc {
  void *req;
  void *rsp;
  u64 req_dma;
  u64 rsp_dma;
  struct ib_sge               sge;
  struct ib_cqe               cqe;
};
```

Members

req pointer to login request buffer

rsp pointer to login response buffer

req_dma DMA address of login request buffer

rsp_dma DMA address of login response buffer

sge IB sge for login post recv

cqe completion handler

struct **iser_comp**
iSER completion context

Definition

```
struct iser_comp {
  struct ib_cq            *cq;
  int active_qps;
};
```

Members

cq completion queue

active_qps Number of active QPs attached to completion context

struct **iser_device**
iSER device handle

Definition

```
struct iser_device {
  struct ib_device             *ib_device;
  struct ib_pd                 *pd;
  struct ib_event_handler      event_handler;
  struct list_head             ig_list;
  int refcount;
  int comps_used;
  struct iser_comp             *comps;
};
```

Members

ib_device RDMA device

pd Protection Domain for this device

event_handler IB events handle routine

ig_list entry in devices list

refcount Reference counter, dominated by open iser connections

comps_used Number of completion contexts used, Min between online cpus and device max completion vectors

comps Dinamically allocated array of completion handlers

struct **iser_reg_resources**
Fast registration resources

Definition

```
struct iser_reg_resources {
  struct ib_mr                    *mr;
  struct ib_mr                    *sig_mr;
  u8 mr_valid:1;
};
```

Members

mr memory region

sig_mr signature memory region

mr_valid is mr valid indicator

struct **iser_fr_desc**
Fast registration descriptor

Definition

```
struct iser_fr_desc {
  struct list_head                list;
  struct iser_reg_resources       rsc;
  bool sig_protected;
  struct list_head                all_list;
};
```

Members

list entry in connection fastreg pool

rsc data buffer registration resources

sig_protected is region protected indicator

struct **iser_fr_pool**
connection fast registration pool

Definition

```
struct iser_fr_pool {
  struct list_head        list;
  spinlock_t lock;
  int size;
  struct list_head        all_list;
};
```

Members

list list of fastreg descriptors

lock protects fastreg pool

size size of the pool

struct **ib_conn**
Infiniband related objects

Definition

```
struct ib_conn {
  struct rdma_cm_id           *cma_id;
  struct ib_qp                *qp;
  int post_recv_buf_count;
  u8 sig_count;
  struct ib_recv_wr            rx_wr[ISER_MIN_POSTED_RX];
  struct iser_device          *device;
  struct iser_comp            *comp;
  struct iser_fr_pool          fr_pool;
  bool pi_support;
  struct ib_cqe                reg_cqe;
};
```

Members

cma_id rdma_cm connection maneger handle

qp Connection Queue-pair

post_recv_buf_count post receive counter

sig_count send work request signal count

rx_wr receive work request for batch posts

device reference to iser device

comp iser completion context

fr_pool connection fast registration poool

pi_support Indicate device T10-PI support

reg_cqe completion handler

struct **iser_conn**
iSER connection context

Definition

```
struct iser_conn {
  struct ib_conn               ib_conn;
  struct iscsi_conn            *iscsi_conn;
  struct iscsi_endpoint        *ep;
  enum iser_conn_state         state;
  unsigned qp_max_recv_dtos;
  unsigned qp_max_recv_dtos_mask;
  unsigned min_posted_rx;
  u16 max_cmds;
  char name[ISER_OBJECT_NAME_SIZE];
  struct work_struct           release_work;
  struct mutex                 state_mutex;
  struct completion            stop_completion;
  struct completion            ib_completion;
  struct completion            up_completion;
  struct list_head             conn_list;
  struct iser_login_desc       login_desc;
  unsigned int                 rx_desc_head;
  struct iser_rx_desc          *rx_descs;
  u32 num_rx_descs;
  unsigned short               scsi_sg_tablesize;
  unsigned short               pages_per_mr;
  bool snd_w_inv;
};
```

Members

ib_conn connection RDMA resources

iscsi_conn link to matching iscsi connection

ep transport handle

state connection logical state

qp_max_recv_dtos maximum number of data outs, corresponds to max number of post recvs

qp_max_recv_dtos_mask (qp_max_recv_dtos - 1)

min_posted_rx (qp_max_recv_dtos >> 2)

max_cmds maximum cmds allowed for this connection

name connection peer portal

release_work deffered work for release job

state_mutex protects iser onnection state

stop_completion conn_stop completion

ib_completion RDMA cleanup completion

up_completion connection establishment completed (state is ISER_CONN_UP)

conn_list entry in ig conn list

login_desc login descriptor

rx_desc_head head of rx_descs cyclic buffer

rx_descs rx buffers array (cyclic buffer)

num_rx_descs number of rx descriptors

scsi_sg_tablesize scsi host sg_tablesize

pages_per_mr maximum pages available for registration

snd_w_inv connection uses remote invalidation

struct **iscsi_iser_task**
iser task context

Definition

```
struct iscsi_iser_task {
  struct iser_tx_desc          desc;
  struct iser_conn             *iser_conn;
  enum iser_task_status        status;
  struct scsi_cmnd             *sc;
  int command_sent;
  int dir[ISER_DIRS_NUM];
  struct iser_mem_reg          rdma_reg[ISER_DIRS_NUM];
  struct iser_data_buf         data[ISER_DIRS_NUM];
  struct iser_data_buf         prot[ISER_DIRS_NUM];
};
```

Members

desc TX descriptor

iser_conn link to iser connection

status current task status

sc link to scsi command

command_sent indicate if command was sent

dir iser data direction

rdma_reg task rdma registration desc

data iser data buffer desc

prot iser protection buffer desc

struct **iser_global**
iSER global context

Definition

```
struct iser_global {
  struct mutex      device_list_mutex;
  struct list_head  device_list;
  struct mutex      connlist_mutex;
  struct list_head  connlist;
  struct kmem_cache *desc_cache;
};
```

Members

device_list_mutex protects device_list

device_list iser devices global list

connlist_mutex protects connlist

connlist iser connections global list

desc_cache kmem cache for tx dataout

int **iscsi_iser_pdu_alloc**(struct iscsi_task * task, uint8_t opcode)
allocate an iscsi-iser PDU

Parameters

struct iscsi_task * task iscsi task

uint8_t opcode iscsi command opcode

Description

Notes: This routine can't fail, just assign iscsi task hdr and max hdr size.

int **iser_initialize_task_headers**(struct iscsi_task * task, struct iser_tx_desc * tx_desc)
Initialize task headers

Parameters

struct iscsi_task * task iscsi task

struct iser_tx_desc * tx_desc iser tx descriptor

Notes

This routine may race with iser teardown flow for scsi error handling TMFs. So for TMF we should acquire the state mutex to avoid dereferencing the IB device which may have already been terminated.

int **iscsi_iser_task_init**(struct iscsi_task * task)
Initialize iscsi-iser task

Parameters

struct iscsi_task * task iscsi task

Description

Initialize the task for the scsi command or mgmt command.

Return

Returns zero on success or -ENOMEM when failing to init task headers (dma mapping error).

int **iscsi_iser_mtask_xmit**(struct iscsi_conn * conn, struct iscsi_task * task)
xmit management (immediate) task

Parameters

struct iscsi_conn * conn iscsi connection

struct iscsi_task * task task management task

Notes

The function can return -EAGAIN in which case caller must call it again later, or recover. '0' return code means successful xmit.

int **iscsi_iser_task_xmit**(struct iscsi_task * task)
xmit iscsi-iser task

Parameters

struct iscsi_task * task iscsi task

Return

zero on success or escalates $error on failure.

void **iscsi_iser_cleanup_task**(struct iscsi_task * task)
cleanup an iscsi-iser task

Parameters

struct iscsi_task * task iscsi task

Notes

In case the RDMA device is already NULL (might have been removed in DEVICE_REMOVAL CM event it will bail-out without doing dma unmapping.

u8 **iscsi_iser_check_protection**(struct iscsi_task * task, sector_t * sector)
check protection information status of task.

Parameters

struct iscsi_task * task iscsi task

sector_t * sector error sector if exsists (output)

Return

zero if no data-integrity errors have occured 0x1: data-integrity error occured in the guard-block 0x2: data-integrity error occured in the reference tag 0x3: data-integrity error occured in the application tag

In addition the error sector is marked.

struct iscsi_cls_conn * **iscsi_iser_conn_create**(struct iscsi_cls_session * cls_session, uint32_t conn_idx)
create a new iscsi-iser connection

Parameters

struct iscsi_cls_session * cls_session iscsi class connection

uint32_t conn_idx connection index within the session (for MCS)

Return

iscsi_cls_conn when iscsi_conn_setup succeeds or NULL otherwise.

int **iscsi_iser_conn_bind**(struct iscsi_cls_session * cls_session, struct iscsi_cls_conn * cls_conn, uint64_t transport_eph, int is_leading)
bind iscsi and iser connection structures

Parameters

struct iscsi_cls_session * cls_session iscsi class session

struct iscsi_cls_conn * cls_conn iscsi class connection

uint64_t transport_eph transport end-point handle

int is_leading indicate if this is the session leading connection (MCS)

Return

zero on success, $error if iscsi_conn_bind fails and -EINVAL in case end-point doesn't exsits anymore or iser connection state is not UP (teardown already started).

int **iscsi_iser_conn_start**(struct iscsi_cls_conn * cls_conn)
start iscsi-iser connection

Parameters

struct iscsi_cls_conn * cls_conn iscsi class connection

Notes

Here iser intialize (or re-initialize) stop_completion as from this point iscsi must call conn_stop in session/connection teardown so iser transport must wait for it.

void **iscsi_iser_conn_stop**(struct iscsi_cls_conn * cls_conn, int flag)
stop iscsi-iser connection

Parameters

struct iscsi_cls_conn * cls_conn iscsi class connection

int flag indicate if recover or terminate (passed as is)

Notes

Calling iscsi_conn_stop might theoretically race with DEVICE_REMOVAL event and dereference a previously freed RDMA device handle, so we call it under iser the state lock to protect against this kind of race.

void **iscsi_iser_session_destroy**(struct iscsi_cls_session * cls_session)
destroy iscsi-iser session

Parameters

struct iscsi_cls_session * cls_session iscsi class session

Description

Removes and free iscsi host.

struct iscsi_cls_session * **iscsi_iser_session_create**(struct iscsi_endpoint * ep, uint16_t cmds_max, uint16_t qdepth, uint32_t initial_cmdsn)

create an iscsi-iser session

Parameters

struct iscsi_endpoint * ep iscsi end-point handle

uint16_t cmds_max maximum commands in this session

uint16_t qdepth session command queue depth

uint32_t initial_cmdsn initiator command sequnce number

Description

Allocates and adds a scsi host, expose DIF supprot if exists, and sets up an iscsi session.

struct iscsi_endpoint * **iscsi_iser_ep_connect**(struct Scsi_Host * shost, struct sockaddr * dst_addr, int non_blocking)

Initiate iSER connection establishment

Parameters

struct Scsi_Host * shost scsi_host

struct sockaddr * dst_addr destination address

int non_blocking indicate if routine can block

Description

Allocate an iscsi endpoint, an iser_conn structure and bind them. After that start RDMA connection establishment via rdma_cm. We don't allocate iser_conn embedded in iscsi_endpoint since in teardown the endpoint will be destroyed at ep_disconnect while iser_conn will cleanup its resources asynchronuously.

Return

iscsi_endpoint created by iscsi layer or ERR_PTR(error) if fails.

int **iscsi_iser_ep_poll**(struct iscsi_endpoint * ep, int timeout_ms)

poll for iser connection establishment to complete

Parameters

struct iscsi_endpoint * ep iscsi endpoint (created at ep_connect)

int timeout_ms polling timeout allowed in ms.

Description

This routine boils down to waiting for up_completion signaling that cma_id got CONNECTED event.

Return

1 if succeeded in connection establishment, 0 if timeout expired (libiscsi will retry will kick in) or -1 if interrupted by signal or more likely iser connection state transitioned to TEMINATING or DOWN during the wait period.

void **iscsi_iser_ep_disconnect**(struct iscsi_endpoint * ep)
Initiate connection teardown process

Parameters

struct iscsi_endpoint * ep iscsi endpoint handle

Description

This routine is not blocked by iser and RDMA termination process completion as we queue a deffered work for iser/RDMA destruction and cleanup or actually call it immediately in case we didn't pass iscsi conn bind/start stage, thus it is safe.

int **iser_send_command**(struct iscsi_conn * conn, struct iscsi_task * task)
send command PDU

Parameters

struct iscsi_conn * conn link to matching iscsi connection

struct iscsi_task * task SCSI command task

int **iser_send_data_out**(struct iscsi_conn * conn, struct iscsi_task * task, struct iscsi_data * hdr)
send data out PDU

Parameters

struct iscsi_conn * conn link to matching iscsi connection

struct iscsi_task * task SCSI command task

struct iscsi_data * hdr pointer to the LLD's iSCSI message header

int **iser_alloc_fastreg_pool**(struct ib_conn * ib_conn, unsigned cmds_max, unsigned int size)
Creates pool of fast_reg descriptors for fast registration work requests.

Parameters

struct ib_conn * ib_conn connection RDMA resources

unsigned cmds_max max number of SCSI commands for this connection

unsigned int size max number of pages per map request

Return

0 on success, or errno code on failure

void **iser_free_fastreg_pool**(struct ib_conn * ib_conn)
releases the pool of fast_reg descriptors

Parameters

struct ib_conn * ib_conn connection RDMA resources

void **iser_free_ib_conn_res**(struct iser_conn * iser_conn, bool destroy)
release IB related resources

Parameters

struct iser_conn * iser_conn iser connection struct

bool destroy indicator if we need to try to release the iser device and memory regoins pool (only iscsi shutdown and DEVICE_REMOVAL will use this).

Description

This routine is called with the iser state mutex held so the cm_id removal is out of here. It is Safe to be invoked multiple times.

void **iser_conn_release**(struct iser_conn * iser_conn)
Frees all conn objects and deallocs conn descriptor

Parameters

struct iser_conn * iser_conn iSER connection context

int **iser_conn_terminate**(struct iser_conn * iser_conn)
triggers start of the disconnect procedures and waits for them to be done

Parameters

struct iser_conn * iser_conn iSER connection context

Description

Called with state mutex held

int **iser_post_send**(struct ib_conn * ib_conn, struct iser_tx_desc * tx_desc, bool signal)
Initiate a Send DTO operation

Parameters

struct ib_conn * ib_conn connection RDMA resources

struct iser_tx_desc * tx_desc iSER TX descriptor

bool signal true to send work request as SIGNALED

Return

0 on success, -1 on failure

14.4.2 Omni-Path (OPA) Virtual NIC support

struct **opa_vnic_ctrl_port**
OPA virtual NIC control port

Definition

```
struct opa_vnic_ctrl_port {
  struct ib_device           *ibdev;
  struct opa_vnic_ctrl_ops   *ops;
  u8 num_ports;
};
```

Members

ibdev pointer to ib device

ops opa vnic control operations

num_ports number of opa ports

struct **opa_vnic_adapter**
OPA VNIC netdev private data structure

Definition

```
struct opa_vnic_adapter {
  struct net_device             *netdev;
  struct ib_device              *ibdev;
  struct opa_vnic_ctrl_port     *cport;
  const struct net_device_ops   *rn_ops;
  u8 port_num;
  u8 vport_num;
  struct mutex lock;
  struct __opa_veswport_info  info;
  u8 vema_mac_addr[ETH_ALEN];
  u32 umac_hash;
  u32 mmac_hash;
  struct hlist_head  __rcu   *mactbl;
  struct mutex mactbl_lock;
  spinlock_t stats_lock;
  u8 flow_tbl[OPA_VNIC_FLOW_TBL_SIZE];
  unsigned long trap_timeout;
  u8 trap_count;
};
```

Members

netdev pointer to associated netdev

ibdev ib device

cport pointer to opa vnic control port

rn_ops rdma netdev’ s net_device_ops

port_num OPA port number

vport_num vesw port number

lock adapter lock

info virtual ethernet switch port information

vema_mac_addr mac address configured by vema

umac_hash unicast maclist hash

mmac_hash multicast maclist hash

mactbl hash table of MAC entries

mactbl_lock mac table lock

stats_lock statistics lock

flow_tbl flow to default port redirection table

trap_timeout trap timeout

trap_count no. of traps allowed within timeout period

struct **opa_vnic_mac_tbl_node**
OPA VNIC mac table node

Definition

```
struct opa_vnic_mac_tbl_node {
  struct hlist_node                    hlist;
  u16 index;
  struct __opa_vnic_mactable_entry     entry;
};
```

Members

hlist hash list handle

index index of entry in the mac table

entry entry in the table

struct **opa_vesw_info**
OPA vnic switch information

Definition

```
struct opa_vesw_info {
  __be16 fabric_id;
  __be16 vesw_id;
  u8 rsvd0[6];
  __be16 def_port_mask;
  u8 rsvd1[2];
  __be16 pkey;
  u8 rsvd2[4];
  __be32 u_mcast_dlid;
  __be32 u_ucast_dlid[OPA_VESW_MAX_NUM_DEF_PORT];
  __be32 rc;
  u8 rsvd3[56];
  __be16 eth_mtu;
  u8 rsvd4[2];
};
```

Members

fabric_id 10-bit fabric id

vesw_id 12-bit virtual ethernet switch id

def_port_mask bitmask of default ports

pkey partition key

u_mcast_dlid unknown multicast dlid

u_ucast_dlid array of unknown unicast dlids

rc routing control

eth_mtu Ethernet MTU

struct **opa_per_veswport_info**
OPA vnic per port information

Definition

```
struct opa_per_veswport_info {
  __be32 port_num;
  u8 eth_link_status;
  u8 rsvd0[3];
  u8 base_mac_addr[ETH_ALEN];
  u8 config_state;
  u8 oper_state;
  __be16 max_mac_tbl_ent;
  __be16 max_smac_ent;
  __be32 mac_tbl_digest;
  u8 rsvd1[4];
  __be32 encap_slid;
  u8 pcp_to_sc_uc[OPA_VNIC_MAX_NUM_PCP];
  u8 pcp_to_vl_uc[OPA_VNIC_MAX_NUM_PCP];
  u8 pcp_to_sc_mc[OPA_VNIC_MAX_NUM_PCP];
  u8 pcp_to_vl_mc[OPA_VNIC_MAX_NUM_PCP];
  u8 non_vlan_sc_uc;
  u8 non_vlan_vl_uc;
  u8 non_vlan_sc_mc;
  u8 non_vlan_vl_mc;
  u8 rsvd2[48];
  __be16 uc_macs_gen_count;
  __be16 mc_macs_gen_count;
  u8 rsvd3[8];
};
```

Members

port_num port number

eth_link_status current ethernet link state

base_mac_addr base mac address

config_state configured port state

oper_state operational port state

max_mac_tbl_ent max number of mac table entries

max_smac_ent max smac entries in mac table

mac_tbl_digest mac table digest

encap_slid base slid for the port

pcp_to_sc_uc sc by pcp index for unicast ethernet packets

pcp_to_vl_uc vl by pcp index for unicast ethernet packets

pcp_to_sc_mc sc by pcp index for multicast ethernet packets

pcp_to_vl_mc vl by pcp index for multicast ethernet packets

non_vlan_sc_uc sc for non-vlan unicast ethernet packets

non_vlan_vl_uc vl for non-vlan unicast ethernet packets

non_vlan_sc_mc sc for non-vlan multicast ethernet packets

non_vlan_vl_mc vl for non-vlan multicast ethernet packets

uc_macs_gen_count generation count for unicast macs list

mc_macs_gen_count generation count for multicast macs list

struct **opa_veswport_info**
OPA vnic port information

Definition

```
struct opa_veswport_info {
  struct opa_vesw_info          vesw;
  struct opa_per_veswport_info  vport;
};
```

Members

vesw OPA vnic switch information

vport OPA vnic per port information

Description

On host, each of the virtual ethernet ports belongs to a different virtual ethernet switches.

struct **opa_veswport_mactable_entry**
single entry in the forwarding table

Definition

```
struct opa_veswport_mactable_entry {
  u8 mac_addr[ETH_ALEN];
  u8 mac_addr_mask[ETH_ALEN];
  __be32 dlid_sd;
};
```

Members

mac_addr MAC address

mac_addr_mask MAC address bit mask

dlid_sd Matching DLID and side data

Description

On the host each virtual ethernet port will have a forwarding table. These tables are used to map a MAC to a LID and other data. For more details see struct opa_veswport_mactable_entries. This is the structure of a single mactable entry

struct **opa_veswport_mactable**
Forwarding table array

Definition

```
struct opa_veswport_mactable {
  __be16 offset;
  __be16 num_entries;
  __be32 mac_tbl_digest;
  struct opa_veswport_mactable_entry  tbl_entries[];
};
```

Members

offset mac table starting offset

num_entries Number of entries to get or set

mac_tbl_digest mac table digest

tbl_entries Array of table entries

Description

The EM sends down this structure in a MAD indicating the starting offset in the forwarding table that this entry is to be loaded into and the number of entries that that this MAD instance contains The mac_tbl_digest has been added to this MAD structure. It will be set by the EM and it will be used by the EM to check if there are any discrepancies with this value and the value maintained by the EM in the case of VNIC port being deleted or unloaded A new instantiation of a VNIC will always have a value of zero. This value is stored as part of the vnic adapter structure and will be accessed by the GET and SET routines for both the mactable entries and the veswport info.

struct **opa_veswport_summary_counters**
summary counters

Definition

```
struct opa_veswport_summary_counters {
  __be16 vp_instance;
  __be16 vesw_id;
  __be32 veswport_num;
  __be64 tx_errors;
  __be64 rx_errors;
  __be64 tx_packets;
  __be64 rx_packets;
  __be64 tx_bytes;
  __be64 rx_bytes;
  __be64 tx_unicast;
  __be64 tx_mcastbcast;
  __be64 tx_untagged;
  __be64 tx_vlan;
  __be64 tx_64_size;
  __be64 tx_65_127;
  __be64 tx_128_255;
  __be64 tx_256_511;
  __be64 tx_512_1023;
  __be64 tx_1024_1518;
  __be64 tx_1519_max;
  __be64 rx_unicast;
  __be64 rx_mcastbcast;
  __be64 rx_untagged;
  __be64 rx_vlan;
  __be64 rx_64_size;
  __be64 rx_65_127;
  __be64 rx_128_255;
  __be64 rx_256_511;
  __be64 rx_512_1023;
  __be64 rx_1024_1518;
  __be64 rx_1519_max;
  __be64 reserved[16];
```

(continues on next page)

(continued from previous page)

```
};
```

Members

vp_instance vport instance on the OPA port

vesw_id virtual ethernet switch id

veswport_num virtual ethernet switch port number

tx_errors transmit errors

rx_errors receive errors

tx_packets transmit packets

rx_packets receive packets

tx_bytes transmit bytes

rx_bytes receive bytes

tx_unicast unicast packets transmitted

tx_mcastbcast multicast/broadcast packets transmitted

tx_untagged non-vlan packets transmitted

tx_vlan vlan packets transmitted

tx_64_size transmit packet length is 64 bytes

tx_65_127 transmit packet length is >=65 and < 127 bytes

tx_128_255 transmit packet length is >=128 and < 255 bytes

tx_256_511 transmit packet length is >=256 and < 511 bytes

tx_512_1023 transmit packet length is >=512 and < 1023 bytes

tx_1024_1518 transmit packet length is >=1024 and < 1518 bytes

tx_1519_max transmit packet length >= 1519 bytes

rx_unicast unicast packets received

rx_mcastbcast multicast/broadcast packets received

rx_untagged non-vlan packets received

rx_vlan vlan packets received

rx_64_size received packet length is 64 bytes

rx_65_127 received packet length is >=65 and < 127 bytes

rx_128_255 received packet length is >=128 and < 255 bytes

rx_256_511 received packet length is >=256 and < 511 bytes

rx_512_1023 received packet length is >=512 and < 1023 bytes

rx_1024_1518 received packet length is >=1024 and < 1518 bytes

rx_1519_max received packet length >= 1519 bytes

Description

All the above are counters of corresponding conditions.

struct **opa_veswport_error_counters**
error counters

Definition

```
struct opa_veswport_error_counters {
  __be16 vp_instance;
  __be16 vesw_id;
  __be32 veswport_num;
  __be64 tx_errors;
  __be64 rx_errors;
  __be64 rsvd0;
  __be64 tx_smac_filt;
  __be64 rsvd1;
  __be64 rsvd2;
  __be64 rsvd3;
  __be64 tx_dlid_zero;
  __be64 rsvd4;
  __be64 tx_logic;
  __be64 rsvd5;
  __be64 tx_drop_state;
  __be64 rx_bad_veswid;
  __be64 rsvd6;
  __be64 rx_runt;
  __be64 rx_oversize;
  __be64 rsvd7;
  __be64 rx_eth_down;
  __be64 rx_drop_state;
  __be64 rx_logic;
  __be64 rsvd8;
  __be64 rsvd9[16];
};
```

Members

vp_instance vport instance on the OPA port

vesw_id virtual ethernet switch id

veswport_num virtual ethernet switch port number

tx_errors transmit errors

rx_errors receive errors

tx_smac_filt smac filter errors

tx_dlid_zero transmit packets with invalid dlid

tx_logic other transmit errors

tx_drop_state packet tansmission in non-forward port state

rx_bad_veswid received packet with invalid vesw id

rx_runt received ethernet packet with length < 64 bytes

rx_oversize received ethernet packet with length > MTU size

rx_eth_down received packets when interface is down

rx_drop_state received packets in non-forwarding port state

rx_logic other receive errors

Description

All the above are counters of corresponding error conditions.

struct **opa_veswport_trap**
Trap message sent to EM by VNIC

Definition

```
struct opa_veswport_trap {
  __be16 fabric_id;
  __be16 veswid;
  __be32 veswportnum;
  __be16 opaportnum;
  u8 veswportindex;
  u8 opcode;
  __be32 reserved;
};
```

Members

fabric_id 10 bit fabric id

veswid 12 bit virtual ethernet switch id

veswportnum logical port number on the Virtual switch

opaportnum physical port num (redundant on host)

veswportindex switch port index on opa port 0 based

opcode operation

reserved 32 bit for alignment

Description

The VNIC will send trap messages to the Ethernet manager to inform it about changes to the VNIC config, behaviour etc. This is the format of the trap payload.

struct **opa_vnic_iface_mac_entry**
single entry in the mac list

Definition

```
struct opa_vnic_iface_mac_entry {
  u8 mac_addr[ETH_ALEN];
};
```

Members

mac_addr MAC address

struct **opa_veswport_iface_macs**
Msg to set globally administered MAC

Definition

```
struct opa_veswport_iface_macs {
  __be16 start_idx;
  __be16 num_macs_in_msg;
  __be16 tot_macs_in_lst;
  __be16 gen_count;
  struct opa_vnic_iface_mac_entry entry[];
};
```

Members

start_idx position of first entry (0 based)

num_macs_in_msg number of MACs in this message

tot_macs_in_lst The total number of MACs the agent has

gen_count gen_count to indicate change

entry The mac list entry

Description

Same attribute IDS and attribute modifiers as in locally administered addresses used to set globally administered addresses

struct **opa_vnic_vema_mad**
Generic VEMA MAD

Definition

```
struct opa_vnic_vema_mad {
  struct ib_mad_hdr  mad_hdr;
  struct ib_rmpp_hdr rmpp_hdr;
  u8 reserved;
  u8 oui[3];
  u8 data[OPA_VNIC_EMA_DATA];
};
```

Members

mad_hdr Generic MAD header

rmpp_hdr RMPP header for vendor specific MADs

oui Unique org identifier

data MAD data

struct **opa_vnic_notice_attr**
Generic Notice MAD

Definition

```
struct opa_vnic_notice_attr {
  u8 gen_type;
  u8 oui_1;
  u8 oui_2;
  u8 oui_3;
  __be16 trap_num;
```

(continues on next page)

(continued from previous page)

```
  __be16 toggle_count;
  __be32 issuer_lid;
  __be32 reserved;
  u8 issuer_gid[16];
  u8 raw_data[64];
};
```

Members

gen_type Generic/Specific bit and type of notice

oui_1 Vendor ID byte 1

oui_2 Vendor ID byte 2

oui_3 Vendor ID byte 3

trap_num Trap number

toggle_count Notice toggle bit and count value

issuer_lid Trap issuer's lid

issuer_gid Issuer GID (only if Report method)

raw_data Trap message body

struct **opa_vnic_vema_mad_trap**
Generic VEMA MAD Trap

Definition

```
struct opa_vnic_vema_mad_trap {
  struct ib_mad_hdr            mad_hdr;
  struct ib_rmpp_hdr           rmpp_hdr;
  u8 reserved;
  u8 oui[3];
  struct opa_vnic_notice_attr  notice;
};
```

Members

mad_hdr Generic MAD header

rmpp_hdr RMPP header for vendor specific MADs

oui Unique org identifier

notice Notice structure

void **opa_vnic_vema_report_event**(struct opa_vnic_adapter * adapter, u8 event)
sent trap to report the specified event

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

u8 event event to be reported

Description

This function calls vema api to sent a trap for the given event.

void **opa_vnic_get_summary_counters**(struct opa_vnic_adapter * adapter, struct opa_veswport_summary_counters * cntrs)
get summary counters

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

struct opa_veswport_summary_counters * cntrs pointer to destination summary counters structure

Description

This function populates the summary counters that is maintained by the given adapter to destination address provided.

void **opa_vnic_get_error_counters**(struct opa_vnic_adapter * adapter, struct opa_veswport_error_counters * cntrs)
get error counters

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

struct opa_veswport_error_counters * cntrs pointer to destination error counters structure

Description

This function populates the error counters that is maintained by the given adapter to destination address provided.

void **opa_vnic_get_vesw_info**(struct opa_vnic_adapter * adapter, struct opa_vesw_info * info)

- Get the vesw information

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

struct opa_vesw_info * info pointer to destination vesw info structure

Description

This function copies the vesw info that is maintained by the given adapter to destination address provided.

void **opa_vnic_set_vesw_info**(struct opa_vnic_adapter * adapter, struct opa_vesw_info * info)

- Set the vesw information

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

struct opa_vesw_info * info pointer to vesw info structure

Description

This function updates the vesw info that is maintained by the given adapter with vesw info provided. Reserved fields are stored and returned back to EM as is.

void **opa_vnic_get_per_veswport_info**(struct opa_vnic_adapter * adapter, struct opa_per_veswport_info * info)

- Get the vesw per port information

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

struct opa_per_veswport_info * info pointer to destination vport info structure

Description

This function copies the vesw per port info that is maintained by the given adapter to destination address provided. Note that the read only fields are not copied.

void **opa_vnic_set_per_veswport_info**(struct opa_vnic_adapter * adapter, struct opa_per_veswport_info * info)

- Set vesw per port information

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

struct opa_per_veswport_info * info pointer to vport info structure

Description

This function updates the vesw per port info that is maintained by the given adapter with vesw per port info provided. Reserved fields are stored and returned back to EM as is.

void **opa_vnic_query_mcast_macs**(struct opa_vnic_adapter * adapter, struct opa_veswport_iface_macs * macs)

query multicast mac list

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

struct opa_veswport_iface_macs * macs pointer mac list

Description

This function populates the provided mac list with the configured multicast addresses in the adapter.

void **opa_vnic_query_ucast_macs**(struct opa_vnic_adapter * adapter, struct opa_veswport_iface_macs * macs)

query unicast mac list

Parameters

struct opa_vnic_adapter * adapter vnic port adapter

struct opa_veswport_iface_macs * macs pointer mac list

Description

This function populates the provided mac list with the configured unicast addresses in the adapter.

struct **opa_vnic_vema_port**

- VNIC VEMA port details

Definition

```
struct opa_vnic_vema_port {
  struct opa_vnic_ctrl_port      *cport;
  struct ib_mad_agent            *mad_agent;
  struct opa_class_port_info      class_port_info;
  u64 tid;
  u8 port_num;
  struct xarray                   vports;
  struct ib_event_handler         event_handler;
  struct mutex                    lock;
};
```

Members

cport pointer to port

mad_agent pointer to mad agent for port

class_port_info Class port info information.

tid Transaction id

port_num OPA port number

vports vnic ports

event_handler ib event handler

lock adapter interface lock

u8 **vema_get_vport_num**(struct opa_vnic_vema_mad * recvd_mad)

- Get the vnic from the mad

Parameters

struct opa_vnic_vema_mad * recvd_mad Received mad

Return

returns value of the vnic port number

struct opa_vnic_adapter * **vema_get_vport_adapter**(struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_port * port)

- Get vnic port adapter from recvd mad

Parameters

struct opa_vnic_vema_mad * recvd_mad received mad

struct opa_vnic_vema_port * port ptr to port struct on which MAD was recvd

Return

vnic adapter

bool **vema_mac_tbl_req_ok**(struct opa_veswport_mactable * mac_tbl)

- Check if mac request has correct values

Parameters

struct opa_veswport_mactable * mac_tbl mac table

Description

This function checks for the validity of the offset and number of entries required.

Return

true if offset and num_entries are valid

struct opa_vnic_adapter * **vema_add_vport**(struct opa_vnic_vema_port * port, u8 vport_num)

- Add a new vnic port

Parameters

struct opa_vnic_vema_port * port ptr to opa_vnic_vema_port struct

u8 vport_num vnic port number (to be added)

Description

Return a pointer to the vnic adapter structure

void **vema_get_class_port_info**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Get class info for port

Parameters

struct opa_vnic_vema_port * port Port on whic MAD was received

struct opa_vnic_vema_mad * recvd_mad pointer to the received mad

struct opa_vnic_vema_mad * rsp_mad pointer to respose mad

Description

This function copies the latest class port info value set for the port and stores it for generating traps

void **vema_set_class_port_info**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Get class info for port

Parameters

struct opa_vnic_vema_port * port Port on whic MAD was received

struct opa_vnic_vema_mad * recvd_mad pointer to the received mad

struct opa_vnic_vema_mad * rsp_mad pointer to respose mad

Description

This function updates the port class info for the specific vnic and sets up the response mad data

void **vema_get_veswport_info**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Get veswport info

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad pointer to the received mad

struct opa_vnic_vema_mad * rsp_mad pointer to respose mad

void **vema_set_veswport_info**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Set veswport info

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad pointer to the received mad

struct opa_vnic_vema_mad * rsp_mad pointer to respose mad

Description

This function gets the port class infor for vnic

void **vema_get_mac_entries**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Get MAC entries in VNIC MAC table

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad pointer to the received mad

struct opa_vnic_vema_mad * rsp_mad pointer to respose mad

Description

This function gets the MAC entries that are programmed into the VNIC MAC forwarding table. It checks for the validity of the index into the MAC table and the number of entries that are to be retrieved.

void **vema_set_mac_entries**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Set MAC entries in VNIC MAC table

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad pointer to the received mad

struct opa_vnic_vema_mad * rsp_mad pointer to respose mad

Description

This function sets the MAC entries in the VNIC forwarding table It checks for the validity of the index and the number of forwarding table entries to be programmed.

void **vema_set_delete_vesw**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Reset VESW info to POD values

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad pointer to the received mad

struct opa_vnic_vema_mad * rsp_mad pointer to respose mad

Description

This function clears all the fields of veswport info for the requested vesw and sets them back to the power-on default values. It does not delete the vesw.

void **vema_get_mac_list**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad, u16 attr_id)

- Get the unicast/multicast macs.

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad Received mad contains fields to set vnic parameters

struct opa_vnic_vema_mad * rsp_mad Response mad to be built

u16 attr_id Attribute ID indicating multicast or unicast mac list

void **vema_get_summary_counters**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Gets summary counters.

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad Received mad contains fields to set vnic parameters

struct opa_vnic_vema_mad * rsp_mad Response mad to be built

void **vema_get_error_counters**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Gets summary counters.

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad Received mad contains fields to set vnic parameters

struct opa_vnic_vema_mad * rsp_mad Response mad to be built

void **vema_get**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Process received get MAD

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad Received mad

struct opa_vnic_vema_mad * rsp_mad Response mad to be built

void **vema_set**(struct opa_vnic_vema_port * port, struct opa_vnic_vema_mad * recvd_mad, struct opa_vnic_vema_mad * rsp_mad)

- Process received set MAD

Parameters

struct opa_vnic_vema_port * port source port on which MAD was received

struct opa_vnic_vema_mad * recvd_mad Received mad contains fields to set vnic parameters

struct opa_vnic_vema_mad * rsp_mad Response mad to be built

void **vema_send**(struct ib_mad_agent * mad_agent, struct ib_mad_send_wc * mad_wc)

- Send handler for VEMA MAD agent

Parameters

struct ib_mad_agent * mad_agent pointer to the mad agent

struct ib_mad_send_wc * mad_wc pointer to mad send work completion information

Description

Free all the data structures associated with the sent MAD

void **vema_recv**(struct ib_mad_agent * mad_agent, struct ib_mad_send_buf * send_buf, struct ib_mad_recv_wc * mad_wc)

- Recv handler for VEMA MAD agent

Parameters

struct ib_mad_agent * mad_agent pointer to the mad agent

struct ib_mad_send_buf * send_buf Send buffer if found, else NULL

struct ib_mad_recv_wc * mad_wc pointer to mad send work completion information

Description

Handle only set and get methods and respond to other methods as unsupported. Allocate response buffer and address handle for the response MAD.

struct opa_vnic_vema_port * **vema_get_port**(struct opa_vnic_ctrl_port * cport, u8 port_num)

- Gets the opa_vnic_vema_port

Parameters

struct opa_vnic_ctrl_port * cport pointer to control dev

u8 port_num Port number

Description

This function loops through the ports and returns the opa_vnic_vema port structure that is associated with the OPA port number

Return

ptr to requested opa_vnic_vema_port strucure if success, NULL if not

void **opa_vnic_vema_send_trap**(struct opa_vnic_adapter * adapter, struct __opa_veswport_trap * data, u32 lid)

- This function sends a trap to the EM

Parameters

struct opa_vnic_adapter * adapter pointer to vnic adapter

struct __opa_veswport_trap * data pointer to trap data filled by calling function

u32 lid issuers lid (encap_slid from vesw_port_info)

Description

This function is called from the VNIC driver to send a trap if there is somethng the EM should be notified about. These events currently are 1) UNICAST INTERFACE MACADDRESS changes 2) MULTICAST INTERFACE MACADDRESS changes 3) ETHERNET LINK STATUS changes While allocating the send mad the remote site qpn used is 1 as this is the well known QP.

void **vema_unregister**(struct opa_vnic_ctrl_port * cport)

- Unregisters agent

Parameters

struct opa_vnic_ctrl_port * cport pointer to control port

Description

This deletes the registration by VEMA for MADs

int **vema_register**(struct opa_vnic_ctrl_port * cport)

- Registers agent

Parameters

struct opa_vnic_ctrl_port * cport pointer to control port

Description

This function registers the handlers for the VEMA MADs

Return

returns 0 on success. non zero otherwise

void **opa_vnic_ctrl_config_dev**(struct opa_vnic_ctrl_port * cport, bool en)

- This function sends a trap to the EM by way of ib_modify_port to indicate support for ethernet on the fabric.

Parameters

struct opa_vnic_ctrl_port * cport pointer to control port

bool en enable or disable ethernet on fabric support

int **opa_vnic_vema_add_one**(struct ib_device * device)

- Handle new ib device

Parameters

struct ib_device * device ib device pointer

Description

Allocate the vnic control port and initialize it.

void **opa_vnic_vema_rem_one**(struct ib_device * device, void * client_data)

- Handle ib device removal

Parameters

struct ib_device * device ib device pointer

void * client_data ib client data

Description

Uninitialize and free the vnic control port.

14.4.3 InfiniBand SCSI RDMA protocol target support

enum **srpt_command_state**
SCSI command state managed by SRPT

Constants

SRPT_STATE_NEW New command arrived and is being processed.

SRPT_STATE_NEED_DATA Processing a write or bidir command and waiting for data arrival.

SRPT_STATE_DATA_IN Data for the write or bidir command arrived and is being processed.

SRPT_STATE_CMD_RSP_SENT SRP_RSP for SRP_CMD has been sent.

SRPT_STATE_MGMT Processing a SCSI task management command.

SRPT_STATE_MGMT_RSP_SENT SRP_RSP for SRP_TSK_MGMT has been sent.

SRPT_STATE_DONE Command processing finished successfully, command processing has been aborted or command processing failed.

struct **srpt_ioctx**
shared SRPT I/O context information

Definition

```
struct srpt_ioctx {
  struct ib_cqe           cqe;
  void *buf;
  dma_addr_t dma;
  uint32_t offset;
  uint32_t index;
};
```

Members

cqe Completion queue element.

buf Pointer to the buffer.

dma DMA address of the buffer.

offset Offset of the first byte in **buf** and **dma** that is actually used.

index Index of the I/O context in its ioctx_ring array.

struct **srpt_recv_ioctx**
SRPT receive I/O context

Definition

```
struct srpt_recv_ioctx {
  struct srpt_ioctx       ioctx;
  struct list_head        wait_list;
  int byte_len;
};
```

Members

ioctx See above.

wait_list Node for insertion in srpt_rdma_ch.cmd_wait_list.

byte_len Number of bytes in **ioctx.buf**.

struct **srpt_send_ioctx**
SRPT send I/O context

Definition

```
struct srpt_send_ioctx {
  struct srpt_ioctx       ioctx;
  struct srpt_rdma_ch     *ch;
  struct srpt_recv_ioctx  *recv_ioctx;
  struct srpt_rw_ctx      s_rw_ctx;
  struct srpt_rw_ctx      *rw_ctxs;
  struct scatterlist      imm_sg;
  struct ib_cqe           rdma_cqe;
  enum srpt_command_state state;
  struct se_cmd           cmd;
  u8 n_rdma;
  u8 n_rw_ctx;
  bool queue_status_only;
  u8 sense_data[TRANSPORT_SENSE_BUFFER];
};
```

Members

ioctx See above.

ch Channel pointer.

recv_ioctx Receive I/O context associated with this send I/O context. Only used for processing immediate data.

s_rw_ctx **rw_ctxs** points here if only a single rw_ctx is needed.

rw_ctxs RDMA read/write contexts.

imm_sg Scatterlist for immediate data.

rdma_cqe RDMA completion queue element.

state I/O context state.

cmd Target core command data structure.

n_rdma Number of work requests needed to transfer this ioctx.

n_rw_ctx Size of rw_ctxs array.

queue_status_only Send a SCSI status back to the initiator but no data.

sense_data Sense data to be sent to the initiator.

enum **rdma_ch_state**
SRP channel state

Constants

CH_CONNECTING QP is in RTR state; waiting for RTU.

CH_LIVE QP is in RTS state.

CH_DISCONNECTING DREQ has been sent and waiting for DREP or DREQ has been received.

CH_DRAINING DREP has been received or waiting for DREP timed out and last work request has been queued.

CH_DISCONNECTED Last completion has been received.

struct **srpt_rdma_ch**
RDMA channel

Definition

```
struct srpt_rdma_ch {
  struct srpt_nexus       *nexus;
  struct ib_qp            *qp;
  union {
    struct {
      struct ib_cm_id         *cm_id;
    } ib_cm;
    struct {
      struct rdma_cm_id       *cm_id;
    } rdma_cm;
  };
  struct ib_cq            *cq;
  struct ib_cqe           zw_cqe;
  struct rcu_head         rcu;
  struct kref             kref;
  struct completion       *closed;
  int rq_size;
  u32 max_rsp_size;
  atomic_t sq_wr_avail;
  struct srpt_port        *sport;
  int max_ti_iu_len;
  atomic_t req_lim;
  atomic_t req_lim_delta;
  u16 imm_data_offset;
  spinlock_t spinlock;
  enum rdma_ch_state      state;
  struct kmem_cache       *rsp_buf_cache;
  struct srpt_send_ioctx  **ioctx_ring;
  struct kmem_cache       *req_buf_cache;
  struct srpt_recv_ioctx  **ioctx_recv_ring;
  struct list_head        list;
  struct list_head        cmd_wait_list;
  uint16_t pkey;
  bool using_rdma_cm;
  bool processing_wait_list;
  struct se_session       *sess;
  u8 sess_name[40];
  struct work_struct      release_work;
};
```

Members

nexus I_T nexus this channel is associated with.

qp IB queue pair used for communicating over this channel.

{unnamed_union} anonymous

ib_cm See below.

ib_cm.cm_id IB CM ID associated with the channel.

rdma_cm See below.

rdma_cm.cm_id RDMA CM ID associated with the channel.

cq IB completion queue for this channel.

zw_cqe Zero-length write CQE.

rcu RCU head.

kref kref for this channel.

closed Completion object that will be signaled as soon as a new channel object with the same identity can be created.

rq_size IB receive queue size.

max_rsp_size Maximum size of an RSP response message in bytes.

sq_wr_avail number of work requests available in the send queue.

sport pointer to the information of the HCA port used by this channel.

max_ti_iu_len maximum target-to-initiator information unit length.

req_lim request limit: maximum number of requests that may be sent by the initiator without having received a response.

req_lim_delta Number of credits not yet sent back to the initiator.

imm_data_offset Offset from start of SRP_CMD for immediate data.

spinlock Protects free_list and state.

state channel state. See also enum rdma_ch_state.

rsp_buf_cache kmem_cache for **ioctx_ring**.

ioctx_ring Send ring.

req_buf_cache kmem_cache for **ioctx_recv_ring**.

ioctx_recv_ring Receive I/O context ring.

list Node in srpt_nexus.ch_list.

cmd_wait_list List of SCSI commands that arrived before the RTU event. This list contains struct srpt_ioctx elements and is protected against concurrent modification by the cm_id spinlock.

pkey P_Key of the IB partition for this SRP channel.

using_rdma_cm Whether the RDMA/CM or IB/CM is used for this channel.

processing_wait_list Whether or not cmd_wait_list is being processed.

sess Session information associated with this SRP channel.

sess_name Session name.

release_work Allows scheduling of srpt_release_channel().

struct **srpt_nexus**
I_T nexus

Definition

```
struct srpt_nexus {
  struct rcu_head         rcu;
  struct list_head        entry;
  struct list_head        ch_list;
  u8 i_port_id[16];
  u8 t_port_id[16];
};
```

Members

rcu RCU head for this data structure.

entry srpt_port.nexus_list list node.

ch_list struct srpt_rdma_ch list. Protected by srpt_port.mutex.

i_port_id 128-bit initiator port identifier copied from SRP_LOGIN_REQ.

t_port_id 128-bit target port identifier copied from SRP_LOGIN_REQ.

struct **srpt_port_attrib**
attributes for SRPT port

Definition

```
struct srpt_port_attrib {
  u32 srp_max_rdma_size;
  u32 srp_max_rsp_size;
  u32 srp_sq_size;
  bool use_srq;
};
```

Members

srp_max_rdma_size Maximum size of SRP RDMA transfers for new connections.

srp_max_rsp_size Maximum size of SRP response messages in bytes.

srp_sq_size Shared receive queue (SRQ) size.

use_srq Whether or not to use SRQ.

struct **srpt_tpg**
information about a single “target portal group”

Definition

```
struct srpt_tpg {
  struct list_head        entry;
  struct srpt_port_id     *sport_id;
  struct se_portal_group  tpg;
};
```

Members

entry Entry in **sport_id->tpg_list**.

sport_id Port name this TPG is associated with.

tpg LIO TPG data structure.

Description

Zero or more target portal groups are associated with each port name (srpt_port_id). With each TPG an ACL list is associated.

struct **srpt_port_id**
information about an RDMA port name

Definition

```
struct srpt_port_id {
  struct mutex            mutex;
  struct list_head        tpg_list;
  struct se_wwn           wwn;
  char name[64];
};
```

Members

mutex Protects **tpg_list** changes.

tpg_list TPGs associated with the RDMA port name.

wwn WWN associated with the RDMA port name.

name ASCII representation of the port name.

Description

Multiple sysfs directories can be associated with a single RDMA port. This data structure represents a single (port, name) pair.

struct **srpt_port**
information associated by SRPT with a single IB port

Definition

```
struct srpt_port {
  struct srpt_device      *sdev;
  struct ib_mad_agent     *mad_agent;
  bool enabled;
  u8 port;
  u32 sm_lid;
  u32 lid;
  union ib_gid            gid;
  struct work_struct      work;
  struct srpt_port_id     port_guid_id;
  struct srpt_port_id     port_gid_id;
  struct srpt_port_attrib port_attrib;
  atomic_t refcount;
  struct completion       *freed_channels;
  struct mutex            mutex;
  struct list_head        nexus_list;
};
```

Members

sdev backpointer to the HCA information.

mad_agent per-port management datagram processing information.

enabled Whether or not this target port is enabled.

port one-based port number.

sm_lid cached value of the port's sm_lid.

lid cached value of the port's lid.

gid cached value of the port's gid.

work work structure for refreshing the aforementioned cached values.

port_guid_id target port GUID

port_gid_id target port GID

port_attrib Port attributes that can be accessed through configfs.

refcount Number of objects associated with this port.

freed_channels Completion that will be signaled once **refcount** becomes 0.

mutex Protects nexus_list.

nexus_list Nexus list. See also srpt_nexus.entry.

struct **srpt_device**
information associated by SRPT with a single HCA

Definition

```
struct srpt_device {
  struct ib_device        *device;
  struct ib_pd            *pd;
  u32 lkey;
  struct ib_srq           *srq;
  struct ib_cm_id         *cm_id;
  int srq_size;
  struct mutex            sdev_mutex;
  bool use_srq;
  struct kmem_cache       *req_buf_cache;
  struct srpt_recv_ioctx  **ioctx_ring;
  struct ib_event_handler event_handler;
  struct list_head        list;
  struct srpt_port        port[];
};
```

Members

device Backpointer to the struct ib_device managed by the IB core.

pd IB protection domain.

lkey L_Key (local key) with write access to all local memory.

srq Per-HCA SRQ (shared receive queue).

cm_id Connection identifier.

srq_size SRQ size.

sdev_mutex Serializes use_srq changes.

use_srq Whether or not to use SRQ.

req_buf_cache kmem_cache for **ioctx_ring** buffers.

ioctx_ring Per-HCA SRQ.

event_handler Per-HCA asynchronous IB event handler.

list Node in srpt_dev_list.

port Information about the ports owned by this HCA.

void **srpt_event_handler**(struct ib_event_handler * handler, struct ib_event * event)
asynchronous IB event callback function

Parameters

struct ib_event_handler * handler IB event handler registered by `ib_register_event_handler()`.

struct ib_event * event Description of the event that occurred.

Description

Callback function called by the InfiniBand core when an asynchronous IB event occurs. This callback may occur in interrupt context. See also section 11.5.2, Set Asynchronous Event Handler in the InfiniBand Architecture Specification.

void **srpt_srq_event**(struct ib_event * event, void * ctx)
SRQ event callback function

Parameters

struct ib_event * event Description of the event that occurred.

void * ctx Context pointer specified at SRQ creation time.

void **srpt_qp_event**(struct ib_event * event, struct srpt_rdma_ch * ch)
QP event callback function

Parameters

struct ib_event * event Description of the event that occurred.

struct srpt_rdma_ch * ch SRPT RDMA channel.

void **srpt_set_ioc**(u8 * c_list, u32 slot, u8 value)
initialize a IOUnitInfo structure

Parameters

u8 * c_list controller list.

u32 slot one-based slot number.

u8 value four-bit value.

Description

Copies the lowest four bits of value in element slot of the array of four bit elements called c_list (controller list). The index slot is one-based.

void **srpt_get_class_port_info**(struct ib_dm_mad * mad)
copy ClassPortInfo to a management datagram

Parameters

struct ib_dm_mad * mad Datagram that will be sent as response to DM_ATTR_CLASS_PORT_INFO.

Description

See also section 16.3.3.1 ClassPortInfo in the InfiniBand Architecture Specification.

void **srpt_get_iou**(struct ib_dm_mad * mad)
write IOUnitInfo to a management datagram

Parameters

struct ib_dm_mad * mad Datagram that will be sent as response to DM_ATTR_IOU_INFO.

Description

See also section 16.3.3.3 IOUnitInfo in the InfiniBand Architecture Specification. See also section B.7, table B.6 in the SRP r16a document.

void **srpt_get_ioc**(struct srpt_port * sport, u32 slot, struct ib_dm_mad * mad)
write IOControllerprofile to a management datagram

Parameters

struct srpt_port * sport HCA port through which the MAD has been received.

u32 slot Slot number specified in DM_ATTR_IOC_PROFILE query.

struct ib_dm_mad * mad Datagram that will be sent as response to DM_ATTR_IOC_PROFILE.

Description

See also section 16.3.3.4 IOControllerProfile in the InfiniBand Architecture Specification. See also section B.7, table B.7 in the SRP r16a document.

void **srpt_get_svc_entries**(u64 ioc_guid, u16 slot, u8 hi, u8 lo, struct ib_dm_mad * mad)
write ServiceEntries to a management datagram

Parameters

u64 ioc_guid I/O controller GUID to use in reply.

u16 slot I/O controller number.

u8 hi End of the range of service entries to be specified in the reply.

u8 lo Start of the range of service entries to be specified in the reply..

struct ib_dm_mad * mad Datagram that will be sent as response to DM_ATTR_SVC_ENTRIES.

Description

See also section 16.3.3.5 ServiceEntries in the InfiniBand Architecture Specification. See also section B.7, table B.8 in the SRP r16a document.

void **srpt_mgmt_method_get**(struct srpt_port * sp, struct ib_mad * rq_mad, struct ib_dm_mad * rsp_mad)
process a received management datagram

Parameters

struct srpt_port * sp HCA port through which the MAD has been received.

struct ib_mad * rq_mad received MAD.

struct ib_dm_mad * rsp_mad response MAD.

void **srpt_mad_send_handler**(struct ib_mad_agent * mad_agent, struct ib_mad_send_wc * mad_wc)
MAD send completion callback

Parameters

struct ib_mad_agent * mad_agent Return value of ib_register_mad_agent().

struct ib_mad_send_wc * mad_wc Work completion reporting that the MAD has been sent.

void **srpt_mad_recv_handler**(struct ib_mad_agent * mad_agent, struct ib_mad_send_buf * send_buf, struct ib_mad_recv_wc * mad_wc)
MAD reception callback function

Parameters

struct ib_mad_agent * mad_agent Return value of ib_register_mad_agent().

struct ib_mad_send_buf * send_buf Not used.

struct ib_mad_recv_wc * mad_wc Work completion reporting that a MAD has been received.

int **srpt_refresh_port**(struct srpt_port * sport)
configure a HCA port

Parameters

struct srpt_port * sport SRPT HCA port.

Description

Enable InfiniBand management datagram processing, update the cached sm_lid, lid and gid values, and register a callback function for processing MADs on the specified port.

Note

It is safe to call this function more than once for the same port.

void **srpt_unregister_mad_agent**(struct srpt_device * sdev)
unregister MAD callback functions

Parameters

struct srpt_device * sdev SRPT HCA pointer.

Note

It is safe to call this function more than once for the same device.

struct srpt_ioctx * **srpt_alloc_ioctx**(struct srpt_device * sdev, int ioctx_size, struct kmem_cache * buf_cache, enum dma_data_direction dir)

allocate a SRPT I/O context structure

Parameters

struct srpt_device * sdev SRPT HCA pointer.

int ioctx_size I/O context size.

struct kmem_cache * buf_cache I/O buffer cache.

enum dma_data_direction dir DMA data direction.

void **srpt_free_ioctx**(struct srpt_device * sdev, struct srpt_ioctx * ioctx, struct kmem_cache * buf_cache, enum dma_data_direction dir)

free a SRPT I/O context structure

Parameters

struct srpt_device * sdev SRPT HCA pointer.

struct srpt_ioctx * ioctx I/O context pointer.

struct kmem_cache * buf_cache I/O buffer cache.

enum dma_data_direction dir DMA data direction.

struct srpt_ioctx ** **srpt_alloc_ioctx_ring**(struct srpt_device * sdev, int ring_size, int ioctx_size, struct kmem_cache * buf_cache, int alignment_offset, enum dma_data_direction dir)

allocate a ring of SRPT I/O context structures

Parameters

struct srpt_device * sdev Device to allocate the I/O context ring for.

int ring_size Number of elements in the I/O context ring.

int ioctx_size I/O context size.

struct kmem_cache * buf_cache I/O buffer cache.

int alignment_offset Offset in each ring buffer at which the SRP information unit starts.

enum dma_data_direction dir DMA data direction.

void **srpt_free_ioctx_ring**(struct srpt_ioctx ** ioctx_ring, struct srpt_device * sdev, int ring_size, struct kmem_cache * buf_cache, enum dma_data_direction dir)

free the ring of SRPT I/O context structures

Parameters

struct srpt_ioctx ** ioctx_ring I/O context ring to be freed.

struct srpt_device * sdev SRPT HCA pointer.

int ring_size Number of ring elements.

struct kmem_cache * buf_cache I/O buffer cache.

enum dma_data_direction dir DMA data direction.

enum srpt_command_state **srpt_set_cmd_state**(struct srpt_send_ioctx * ioctx, enum srpt_command_state new)
set the state of a SCSI command

Parameters

struct srpt_send_ioctx * ioctx Send I/O context.

enum srpt_command_state new New I/O context state.

Description

Does not modify the state of aborted commands. Returns the previous command state.

bool **srpt_test_and_set_cmd_state**(struct srpt_send_ioctx * ioctx, enum srpt_command_state old, enum srpt_command_state new)
test and set the state of a command

Parameters

struct srpt_send_ioctx * ioctx Send I/O context.

enum srpt_command_state old Current I/O context state.

enum srpt_command_state new New I/O context state.

Description

Returns true if and only if the previous command state was equal to ‘old’.

int **srpt_post_recv**(struct srpt_device * sdev, struct srpt_rdma_ch * ch, struct srpt_recv_ioctx * ioctx)
post an IB receive request

Parameters

struct srpt_device * sdev SRPT HCA pointer.

struct srpt_rdma_ch * ch SRPT RDMA channel.

struct srpt_recv_ioctx * ioctx Receive I/O context pointer.

int **srpt_zerolength_write**(struct srpt_rdma_ch * ch)
perform a zero-length RDMA write

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

Description

A quote from the InfiniBand specification: C9-88: For an HCA responder using Reliable Connection service, for each zero-length RDMA READ or WRITE request, the R_Key shall not be validated, even if the request includes Immediate data.

int **srpt_get_desc_tbl**(struct srpt_recv_ioctx * recv_ioctx, struct srpt_send_ioctx * ioctx, struct srp_cmd * srp_cmd, enum dma_data_direction * dir, struct scatterlist ** sg, unsigned int * sg_cnt, u64 * data_len, u16 imm_data_offset)
parse the data descriptors of a SRP_CMD request

Parameters

struct srpt_recv_ioctx * recv_ioctx I/O context associated with the received command **srp_cmd**.

struct srpt_send_ioctx * ioctx I/O context that will be used for responding to the initiator.

struct srp_cmd * srp_cmd Pointer to the SRP_CMD request data.

enum dma_data_direction * dir Pointer to the variable to which the transfer direction will be written.

struct scatterlist ** sg [out] scatterlist for the parsed SRP_CMD.

unsigned int * sg_cnt [out] length of **sg**.

u64 * data_len Pointer to the variable to which the total data length of all descriptors in the SRP_CMD request will be written.

u16 imm_data_offset [in] Offset in SRP_CMD requests at which immediate data starts.

Description

This function initializes ioctx->nrbuf and ioctx->r_bufs.

Returns -EINVAL when the SRP_CMD request contains inconsistent descriptors; -ENOMEM when memory allocation fails and zero upon success.

int **srpt_init_ch_qp**(struct srpt_rdma_ch * ch, struct ib_qp * qp)
initialize queue pair attributes

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

struct ib_qp * qp Queue pair pointer.

Description

Initialized the attributes of queue pair 'qp' by allowing local write, remote read and remote write. Also transitions 'qp' to state IB_QPS_INIT.

int **srpt_ch_qp_rtr**(struct srpt_rdma_ch * ch, struct ib_qp * qp)
change the state of a channel to 'ready to receive' (RTR)

Parameters

struct srpt_rdma_ch * ch channel of the queue pair.

struct ib_qp * qp queue pair to change the state of.

Description

Returns zero upon success and a negative value upon failure.

Note

currently a struct ib_qp_attr takes 136 bytes on a 64-bit system. If this structure ever becomes larger, it might be necessary to allocate it dynamically instead of on the stack.

int **srpt_ch_qp_rts**(struct srpt_rdma_ch * ch, struct ib_qp * qp)
change the state of a channel to 'ready to send' (RTS)

Parameters

struct srpt_rdma_ch * ch channel of the queue pair.

struct ib_qp * qp queue pair to change the state of.

Description

Returns zero upon success and a negative value upon failure.

Note

currently a struct ib_qp_attr takes 136 bytes on a 64-bit system. If this structure ever becomes larger, it might be necessary to allocate it dynamically instead of on the stack.

int **srpt_ch_qp_err**(struct srpt_rdma_ch * ch)
set the channel queue pair state to 'error'

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

struct srpt_send_ioctx * **srpt_get_send_ioctx**(struct srpt_rdma_ch * ch)
obtain an I/O context for sending to the initiator

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

int **srpt_abort_cmd**(struct srpt_send_ioctx * ioctx)
abort a SCSI command

Parameters

struct srpt_send_ioctx * ioctx I/O context associated with the SCSI command.

void **srpt_rdma_read_done**(struct ib_cq * cq, struct ib_wc * wc)
RDMA read completion callback

Parameters

struct ib_cq * cq Completion queue.

struct ib_wc * wc Work completion.

Description

XXX: what is now target_execute_cmd used to be asynchronous, and unmapping the data that has been transferred via IB RDMA had to be postponed until the check_stop_free() callback. None of this is necessary anymore and needs to be cleaned up.

int **srpt_build_cmd_rsp**(struct srpt_rdma_ch * ch, struct srpt_send_ioctx * ioctx, u64 tag, int status)
build a SRP_RSP response

Parameters

struct srpt_rdma_ch * ch RDMA channel through which the request has been received.

struct srpt_send_ioctx * ioctx I/O context associated with the SRP_CMD request. The response will be built in the buffer ioctx->buf points at and hence this function will overwrite the request data.

u64 tag tag of the request for which this response is being generated.

int status value for the STATUS field of the SRP_RSP information unit.

Description

Returns the size in bytes of the SRP_RSP response.

An SRP_RSP response contains a SCSI status or service response. See also section 6.9 in the SRP r16a document for the format of an SRP_RSP response. See also SPC-2 for more information about sense data.

int **srpt_build_tskmgmt_rsp**(struct srpt_rdma_ch * ch, struct srpt_send_ioctx * ioctx, u8 rsp_code, u64 tag)
build a task management response

Parameters

struct srpt_rdma_ch * ch RDMA channel through which the request has been received.

struct srpt_send_ioctx * ioctx I/O context in which the SRP_RSP response will be built.

u8 rsp_code RSP_CODE that will be stored in the response.

u64 tag Tag of the request for which this response is being generated.

Description

Returns the size in bytes of the SRP_RSP response.

An SRP_RSP response contains a SCSI status or service response. See also section 6.9 in the SRP r16a document for the format of an SRP_RSP response.

void **srpt_handle_cmd**(struct srpt_rdma_ch * ch, struct srpt_recv_ioctx * recv_ioctx, struct srpt_send_ioctx * send_ioctx)
process a SRP_CMD information unit

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

struct srpt_recv_ioctx * recv_ioctx Receive I/O context.

struct srpt_send_ioctx * send_ioctx Send I/O context.

void **srpt_handle_tsk_mgmt**(struct srpt_rdma_ch * ch, struct srpt_recv_ioctx * recv_ioctx, struct srpt_send_ioctx * send_ioctx)
process a SRP_TSK_MGMT information unit

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

struct srpt_recv_ioctx * recv_ioctx Receive I/O context.

struct srpt_send_ioctx * send_ioctx Send I/O context.

Description

Returns 0 if and only if the request will be processed by the target core.

For more information about SRP_TSK_MGMT information units, see also section 6.7 in the SRP r16a document.

bool **srpt_handle_new_iu**(struct srpt_rdma_ch * ch, struct srpt_recv_ioctx * recv_ioctx)
process a newly received information unit

Parameters

struct srpt_rdma_ch * ch RDMA channel through which the information unit has been received.

struct srpt_recv_ioctx * recv_ioctx Receive I/O context associated with the information unit.

void **srpt_send_done**(struct ib_cq * cq, struct ib_wc * wc)
send completion callback

Parameters

struct ib_cq * cq Completion queue.

struct ib_wc * wc Work completion.

Note

Although this has not yet been observed during tests, at least in theory it is possible that the `srpt_get_send_ioctx()` call invoked by `srpt_handle_new_iu()` fails. This is possible because the req_lim_delta value in each response is set to one, and it is possible that this response makes the initiator send a new request before the send completion for that response has been processed. This could e.g. happen if the call to srpt_put_send_iotcx() is delayed because of a higher priority interrupt or if IB retransmission causes generation of the send completion to be delayed. Incoming information units for which `srpt_get_send_ioctx()` fails are queued on cmd_wait_list. The code below processes these delayed requests one at a time.

int **srpt_create_ch_ib**(struct srpt_rdma_ch * ch)
create receive and send completion queues

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

bool **srpt_close_ch**(struct srpt_rdma_ch * ch)
close a RDMA channel

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

Description

Make sure all resources associated with the channel will be deallocated at an appropriate time.

Returns true if and only if the channel state has been modified into CH_DRAINING.

int **srpt_cm_req_recv**(struct srpt_device *const sdev, struct ib_cm_id * ib_cm_id, struct rdma_cm_id * rdma_cm_id, u8 port_num, __be16 pkey, const struct srp_login_req * req, const char * src_addr)
process the event IB_CM_REQ_RECEIVED

Parameters

struct srpt_device *const sdev HCA through which the login request was received.

struct ib_cm_id * ib_cm_id IB/CM connection identifier in case of IB/CM.

struct rdma_cm_id * rdma_cm_id RDMA/CM connection identifier in case of RDMA/CM.

u8 port_num Port through which the REQ message was received.

__be16 pkey P_Key of the incoming connection.

const struct srp_login_req * req SRP login request.

const char * src_addr GID (IB/CM) or IP address (RDMA/CM) of the port that submitted the login request.

Description

Ownership of the cm_id is transferred to the target session if this function returns zero. Otherwise the caller remains the owner of cm_id.

void **srpt_cm_rtu_recv**(struct srpt_rdma_ch * ch)
process an IB_CM_RTU_RECEIVED or USER_ESTABLISHED event

Parameters

struct srpt_rdma_ch * ch SRPT RDMA channel.

Description

An RTU (ready to use) message indicates that the connection has been established and that the recipient may begin transmitting.

int **srpt_cm_handler**(struct ib_cm_id * cm_id, const struct ib_cm_event * event)
IB connection manager callback function

Parameters

struct ib_cm_id * cm_id IB/CM connection identifier.

const struct ib_cm_event * event IB/CM event.

Description

A non-zero return value will cause the caller destroy the CM ID.

Note

`srpt_cm_handler()` must only return a non-zero value when transferring ownership of the cm_id to a channel by `srpt_cm_req_recv()` failed. Returning a non-zero value in any other case will trigger a race with the ib_destroy_cm_id() call in srpt_release_channel().

void **srpt_queue_response**(struct se_cmd * cmd)
transmit the response to a SCSI command

Parameters

struct se_cmd * cmd SCSI target command.

Description

Callback function called by the TCM core. Must not block since it can be invoked on the context of the IB completion handler.

int **srpt_release_sport**(struct srpt_port * sport)
disable login and wait for associated channels

Parameters

struct srpt_port * sport SRPT HCA port.

int **srpt_add_one**(struct ib_device * device)
InfiniBand device addition callback function

Parameters

struct ib_device * device Describes a HCA.

void **srpt_remove_one**(struct ib_device * device, void * client_data)
InfiniBand device removal callback function

Parameters

struct ib_device * device Describes a HCA.

void * client_data The value passed as the third argument to `ib_set_client_data()`.

void **srpt_close_session**(struct se_session * se_sess)
forcibly close a session

Parameters

struct se_session * se_sess SCSI target session.

Description

Callback function invoked by the TCM core to clean up sessions associated with a node ACL when the user invokes rmdir /sys/kernel/config/target/$driver/$port/$tpg/acls/$i_port_id

u32 **srpt_sess_get_index**(struct se_session * se_sess)
return the value of scsiAttIntrPortIndex (SCSI-MIB)

Parameters

struct se_session * se_sess SCSI target session.

Description

A quote from RFC 4455 (SCSI-MIB) about this MIB object: This object represents an arbitrary integer used to uniquely identify a particular attached remote initiator port to a particular SCSI target port within a particular SCSI target device within a particular SCSI instance.

int **srpt_parse_i_port_id**(u8 i_port_id, const char * name)
parse an initiator port ID

Parameters

u8 i_port_id Binary 128-bit port ID.

const char * name ASCII representation of a 128-bit initiator port ID.

struct se_portal_group * **srpt_make_tpg**(struct se_wwn * wwn, const char * name)
configfs callback invoked for mkdir /sys/kernel/config/target/$driver/$port/$tpg

Parameters

struct se_wwn * wwn Corresponds to $driver/$port.

const char * name $tpg.

void **srpt_drop_tpg**(struct se_portal_group * tpg)
configfs callback invoked for rmdir /sys/kernel/config/target/$driver/$port/$tpg

Parameters

struct se_portal_group * tpg Target portal group to deregister.

struct se_wwn * **srpt_make_tport**(struct target_fabric_configfs * tf, struct config_group * group, const char * name)
configfs callback invoked for mkdir /sys/kernel/config/target/$driver/$port

Parameters

struct target_fabric_configfs * tf Not used.

struct config_group * group Not used.

const char * name $port.

void **srpt_drop_tport**(struct se_wwn * wwn)
configfs callback invoked for rmdir /sys/kernel/config/target/$driver/$port

Parameters

struct se_wwn * wwn $port.

int **srpt_init_module**(void)
kernel module initialization

Parameters

void no arguments

Note

Since `ib_register_client()` registers callback functions, and since at least one of these callback functions (`srpt_add_one()`) calls target core functions, this driver must be registered with the target core before `ib_register_client()` is called.

14.4.4 iSCSI Extensions for RDMA (iSER) target support

void **isert_conn_terminate**(struct isert_conn * isert_conn)
Initiate connection termination

Parameters

struct isert_conn * isert_conn isert connection struct

Notes

In case the connection state is BOUND, move state to TEMINATING and start teardown sequence (rdma_disconnect). In case the connection state is UP, complete flush as well.

Description

This routine must be called with mutex held. Thus it is safe to call multiple times.

void **isert_put_unsol_pending_cmds**(struct iscsi_conn * conn)
Drop commands waiting for unsolicitate dataout

Parameters

struct iscsi_conn * conn iscsi connection

Description

We might still have commands that are waiting for unsolicited dataouts messages. We must put the extra reference on those before blocking on the target_wait_for_session_cmds

CHAPTER

FIFTEEN

SOUND DEVICES

void **snd_card_unref**(struct snd_card * card)
Unreference the card object

Parameters

struct snd_card * card the card object to unreference

Description

Call this function for the card object that was obtained via `snd_card_ref()` or `snd_lookup_minor_data()`.

snd_printk(fmt, ⋯)
printk wrapper

Parameters

fmt format string

... variable arguments

Description

Works like `printk()` but prints the file and the line of the caller when configured with CONFIG_SND_VERBOSE_PRINTK.

snd_printd(fmt, ⋯)
debug printk

Parameters

fmt format string

... variable arguments

Description

Works like `snd_printk()` for debugging purposes. Ignored when CONFIG_SND_DEBUG is not set.

snd_BUG()
give a BUG warning message and stack trace

Parameters

Description

Calls WARN() if CONFIG_SND_DEBUG is set. Ignored when CONFIG_SND_DEBUG is not set.

snd_printd_ratelimit()

Parameters

snd_BUG_ON(cond)
debugging check macro

Parameters

cond condition to evaluate

Description

Has the same behavior as WARN_ON when CONFIG_SND_DEBUG is set, otherwise just evaluates the conditional and returns the value.

snd_printdd(format, ⋯)
debug printk

Parameters

format format string

... variable arguments

Description

Works like `snd_printk()` for debugging purposes. Ignored when CONFIG_SND_DEBUG_VERBOSE is not set.

int **register_sound_special_device**(const struct file_operations * fops, int unit, struct device * dev)
register a special sound node

Parameters

const struct file_operations * fops File operations for the driver

int unit Unit number to allocate

struct device * dev device pointer

Allocate a special sound device by minor number from the sound subsystem.

Return

The allocated number is returned on success. On failure, a negative error code is returned.

int **register_sound_mixer**(const struct file_operations * fops, int dev)
register a mixer device

Parameters

const struct file_operations * fops File operations for the driver

int dev Unit number to allocate

Allocate a mixer device. Unit is the number of the mixer requested. Pass -1 to request the next free mixer unit.

Return

On success, the allocated number is returned. On failure, a negative error code is returned.

int **register_sound_dsp**(const struct file_operations * fops, int dev)
register a DSP device

Parameters

const struct file_operations * fops File operations for the driver

int dev Unit number to allocate

Allocate a DSP device. Unit is the number of the DSP requested. Pass -1 to request the next free DSP unit.

This function allocates both the audio and dsp device entries together and will always allocate them as a matching pair - eg dsp3/audio3

Return

On success, the allocated number is returned. On failure, a negative error code is returned.

void **unregister_sound_special**(int unit)
unregister a special sound device

Parameters

int unit unit number to allocate

Release a sound device that was allocated with register_sound_special(). The unit passed is the return value from the register function.

void **unregister_sound_mixer**(int unit)
unregister a mixer

Parameters

int unit unit number to allocate

Release a sound device that was allocated with `register_sound_mixer()`. The unit passed is the return value from the register function.

void **unregister_sound_dsp**(int unit)
unregister a DSP device

Parameters

int unit unit number to allocate

Release a sound device that was allocated with `register_sound_dsp()`. The unit passed is the return value from the register function.

Both of the allocated units are released together automatically.

int **snd_pcm_stream_linked**(struct snd_pcm_substream * substream)
Check whether the substream is linked with others

Parameters

struct snd_pcm_substream * substream substream to check

Description

Returns true if the given substream is being linked with others.

snd_pcm_stream_lock_irqsave(substream, flags)
Lock the PCM stream

Parameters

substream PCM substream

flags irq flags

Description

This locks the PCM stream like `snd_pcm_stream_lock()` but with the local IRQ (only when nonatomic is false). In nonatomic case, this is identical as `snd_pcm_stream_lock()`.

snd_pcm_group_for_each_entry(s, substream)
iterate over the linked substreams

Parameters

s the iterator

substream the substream

Description

Iterate over the all linked substreams to the given **substream**. When **substream** isn't linked with any others, this gives returns **substream** itself once.

int **snd_pcm_running**(struct snd_pcm_substream * substream)
Check whether the substream is in a running state

Parameters

struct snd_pcm_substream * substream substream to check

Description

Returns true if the given substream is in the state RUNNING, or in the state DRAINING for playback.

ssize_t **bytes_to_samples**(struct snd_pcm_runtime * runtime, ssize_t size)
Unit conversion of the size from bytes to samples

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

ssize_t size size in bytes

snd_pcm_sframes_t **bytes_to_frames**(struct snd_pcm_runtime * runtime,
ssize_t size)
Unit conversion of the size from bytes to frames

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

ssize_t size size in bytes

ssize_t **samples_to_bytes**(struct snd_pcm_runtime * runtime, ssize_t size)
Unit conversion of the size from samples to bytes

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

ssize_t size size in samples

ssize_t **frames_to_bytes**(struct snd_pcm_runtime * runtime, snd_pcm_sframes_t size)
Unit conversion of the size from frames to bytes

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_sframes_t size size in frames

int **frame_aligned**(struct snd_pcm_runtime * runtime, ssize_t bytes)
Check whether the byte size is aligned to frames

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

ssize_t bytes size in bytes

size_t **snd_pcm_lib_buffer_bytes**(struct snd_pcm_substream * substream)
Get the buffer size of the current PCM in bytes

Parameters

struct snd_pcm_substream * substream PCM substream

size_t **snd_pcm_lib_period_bytes**(struct snd_pcm_substream * substream)
Get the period size of the current PCM in bytes

Parameters

struct snd_pcm_substream * substream PCM substream

snd_pcm_uframes_t **snd_pcm_playback_avail**(struct snd_pcm_runtime * runtime)
Get the available (writable) space for playback

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

Description

Result is between 0 …(boundary - 1)

snd_pcm_uframes_t **snd_pcm_capture_avail**(struct snd_pcm_runtime * runtime)
Get the available (readable) space for capture

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

Description

Result is between 0 …(boundary - 1)

snd_pcm_sframes_t **snd_pcm_playback_hw_avail**(struct snd_pcm_runtime * runtime)
Get the queued space for playback

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_sframes_t **snd_pcm_capture_hw_avail**(struct snd_pcm_runtime * runtime)
Get the free space for capture

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

int **snd_pcm_playback_ready**(struct snd_pcm_substream * substream)
check whether the playback buffer is available

Parameters

struct snd_pcm_substream * substream the pcm substream instance

Description

Checks whether enough free space is available on the playback buffer.

Return

Non-zero if available, or zero if not.

int **snd_pcm_capture_ready**(struct snd_pcm_substream * substream)
check whether the capture buffer is available

Parameters

struct snd_pcm_substream * substream the pcm substream instance

Description

Checks whether enough capture data is available on the capture buffer.

Return

Non-zero if available, or zero if not.

int **snd_pcm_playback_data**(struct snd_pcm_substream * substream)
check whether any data exists on the playback buffer

Parameters

struct snd_pcm_substream * substream the pcm substream instance

Description

Checks whether any data exists on the playback buffer.

Return

Non-zero if any data exists, or zero if not. If stop_threshold is bigger or equal to boundary, then this function returns always non-zero.

int **snd_pcm_playback_empty**(struct snd_pcm_substream * substream)
check whether the playback buffer is empty

Parameters

struct snd_pcm_substream * substream the pcm substream instance

Description

Checks whether the playback buffer is empty.

Return

Non-zero if empty, or zero if not.

int **snd_pcm_capture_empty**(struct snd_pcm_substream * substream)
check whether the capture buffer is empty

Parameters

struct snd_pcm_substream * substream the pcm substream instance

Description

Checks whether the capture buffer is empty.

Return

Non-zero if empty, or zero if not.

void **snd_pcm_trigger_done**(struct snd_pcm_substream * substream, struct snd_pcm_substream * master)
Mark the master substream

Parameters

struct snd_pcm_substream * substream the pcm substream instance

struct snd_pcm_substream * master the linked master substream

Description

When multiple substreams of the same card are linked and the hardware supports the single-shot operation, the driver calls this in the loop in `snd_pcm_group_for_each_entry()` for marking the substream as “done”. Then most of trigger operations are performed only to the given master substream.

The trigger_master mark is cleared at timestamp updates at the end of trigger operations.

unsigned int **params_channels**(const struct snd_pcm_hw_params * p)
Get the number of channels from the hw params

Parameters

const struct snd_pcm_hw_params * p hw params

unsigned int **params_rate**(const struct snd_pcm_hw_params * p)
Get the sample rate from the hw params

Parameters

const struct snd_pcm_hw_params * p hw params

unsigned int **params_period_size**(const struct snd_pcm_hw_params * p)
Get the period size (in frames) from the hw params

Parameters

const struct snd_pcm_hw_params * p hw params

unsigned int **params_periods**(const struct snd_pcm_hw_params * p)
Get the number of periods from the hw params

Parameters

const struct snd_pcm_hw_params * p hw params

unsigned int **params_buffer_size**(const struct snd_pcm_hw_params * p)
Get the buffer size (in frames) from the hw params

Parameters

const struct snd_pcm_hw_params * p hw params

unsigned int **params_buffer_bytes**(const struct snd_pcm_hw_params * p)
Get the buffer size (in bytes) from the hw params

Parameters

const struct snd_pcm_hw_params * p hw params

int **snd_pcm_hw_constraint_single**(struct snd_pcm_runtime * runtime, snd_pcm_hw_param_t var, unsigned int val)
Constrain parameter to a single value

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_hw_param_t var The hw_params variable to constrain

unsigned int val The value to constrain to

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_format_cpu_endian**(snd_pcm_format_t format)
Check the PCM format is CPU-endian

Parameters

snd_pcm_format_t format the format to check

Return

1 if the given PCM format is CPU-endian, 0 if opposite, or a negative error code if endian not specified.

void **snd_pcm_set_runtime_buffer**(struct snd_pcm_substream * substream, struct snd_dma_buffer * bufp)
Set the PCM runtime buffer

Parameters

struct snd_pcm_substream * substream PCM substream to set

struct snd_dma_buffer * bufp the buffer information, NULL to clear

Description

Copy the buffer information to runtime->dma_buffer when **bufp** is non-NULL. Otherwise it clears the current buffer information.

void **snd_pcm_gettime**(struct snd_pcm_runtime * runtime, struct timespec64 * tv)
Fill the timespec64 depending on the timestamp mode

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

struct timespec64 * tv timespec64 to fill

int **snd_pcm_lib_alloc_vmalloc_buffer**(struct snd_pcm_substream * substream, size_t size)
allocate virtual DMA buffer

Parameters

struct snd_pcm_substream * substream the substream to allocate the buffer to

size_t size the requested buffer size, in bytes

Description

Allocates the PCM substream buffer using vmalloc(), i.e., the memory is contiguous in kernel virtual space, but not in physical memory. Use this if the buffer is accessed by kernel code but not by device DMA.

Return

1 if the buffer was changed, 0 if not changed, or a negative error code.

int **snd_pcm_lib_alloc_vmalloc_32_buffer**(struct snd_pcm_substream * substream, size_t size)
allocate 32-bit-addressable buffer

Parameters

struct snd_pcm_substream * substream the substream to allocate the buffer to

size_t size the requested buffer size, in bytes

Description

This function works like `snd_pcm_lib_alloc_vmalloc_buffer()`, but uses vmalloc_32(), i.e., the pages are allocated from 32-bit-addressable memory.

Return

1 if the buffer was changed, 0 if not changed, or a negative error code.

dma_addr_t **snd_pcm_sgbuf_get_addr**(struct snd_pcm_substream * substream, unsigned int ofs)
Get the DMA address at the corresponding offset

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned int ofs byte offset

void * **snd_pcm_sgbuf_get_ptr**(struct snd_pcm_substream * substream, unsigned int ofs)
Get the virtual address at the corresponding offset

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned int ofs byte offset

unsigned int **snd_pcm_sgbuf_get_chunk_size**(struct snd_pcm_substream * substream, unsigned int ofs, unsigned int size)
Compute the max size that fits within the contig. page from the given size

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned int ofs byte offset

unsigned int size byte size to examine

void **snd_pcm_mmap_data_open**(struct vm_area_struct * area)
increase the mmap counter

Parameters

struct vm_area_struct * area VMA

Description

PCM mmap callback should handle this counter properly

void **snd_pcm_mmap_data_close**(struct vm_area_struct * area)
decrease the mmap counter

Parameters

struct vm_area_struct * area VMA

Description

PCM mmap callback should handle this counter properly

void **snd_pcm_limit_isa_dma_size**(int dma, size_t * max)
Get the max size fitting with ISA DMA transfer

Parameters

int dma DMA number

size_t * max pointer to store the max size

const char * **snd_pcm_stream_str**(struct snd_pcm_substream * substream)
Get a string naming the direction of a stream

Parameters

struct snd_pcm_substream * substream the pcm substream instance

Return

A string naming the direction of the stream.

struct snd_pcm_substream * **snd_pcm_chmap_substream**(struct snd_pcm_chmap * info, unsigned int idx)
get the PCM substream assigned to the given chmap info

Parameters

struct snd_pcm_chmap * info chmap information

unsigned int idx the substream number index

u64 **pcm_format_to_bits**(snd_pcm_format_t pcm_format)
Strong-typed conversion of pcm_format to bitwise

Parameters

snd_pcm_format_t pcm_format PCM format

pcm_for_each_format(f)
helper to iterate for each format type

Parameters

f the iterator variable in snd_pcm_format_t type

const char * **snd_pcm_format_name**(snd_pcm_format_t format)
Return a name string for the given PCM format

Parameters

snd_pcm_format_t format PCM format

int **snd_pcm_new_stream**(struct snd_pcm * pcm, int stream, int substream_count)
create a new PCM stream

Parameters

struct snd_pcm * pcm the pcm instance

int stream the stream direction, SNDRV_PCM_STREAM_XXX

int substream_count the number of substreams

Description

Creates a new stream for the pcm. The corresponding stream on the pcm must have been empty before calling this, i.e. zero must be given to the argument of `snd_pcm_new()`.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_new**(struct snd_card * card, const char * id, int device, int playback_count, int capture_count, struct snd_pcm ** rpcm)
create a new PCM instance

Parameters

struct snd_card * card the card instance

const char * id the id string

int device the device index (zero based)

int playback_count the number of substreams for playback

int capture_count the number of substreams for capture

struct snd_pcm ** rpcm the pointer to store the new pcm instance

Description

Creates a new PCM instance.

The pcm operators have to be set afterwards to the new instance via `snd_pcm_set_ops()`.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_new_internal**(struct snd_card * card, const char * id, int device, int playback_count, int capture_count, struct snd_pcm ** rpcm)
create a new internal PCM instance

Parameters

struct snd_card * card the card instance

const char * id the id string

int device the device index (zero based - shared with normal PCMs)

int playback_count the number of substreams for playback

int capture_count the number of substreams for capture

struct snd_pcm ** rpcm the pointer to store the new pcm instance

Description

Creates a new internal PCM instance with no userspace device or procfs entries. This is used by ASoC Back End PCMs in order to create a PCM that will only be used internally by kernel drivers. i.e. it cannot be opened by userspace. It provides existing ASoC components drivers with a substream and access to any private data.

The pcm operators have to be set afterwards to the new instance via `snd_pcm_set_ops()`.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_notify**(struct snd_pcm_notify * notify, int nfree)
Add/remove the notify list

Parameters

struct snd_pcm_notify * notify PCM notify list

int nfree 0 = register, 1 = unregister

Description

This adds the given notifier to the global list so that the callback is called for each registered PCM devices. This exists only for PCM OSS emulation, so far.

int **snd_device_new**(struct snd_card * card, enum snd_device_type type, void * device_data, const struct snd_device_ops * ops)
create an ALSA device component

Parameters

struct snd_card * card the card instance

enum snd_device_type type the device type, SNDRV_DEV_XXX

void * device_data the data pointer of this device

const struct snd_device_ops * ops the operator table

Description

Creates a new device component for the given data pointer. The device will be assigned to the card and managed together by the card.

The data pointer plays a role as the identifier, too, so the pointer address must be unique and unchanged.

Return

Zero if successful, or a negative error code on failure.

void **snd_device_disconnect**(struct snd_card * card, void * device_data)
disconnect the device

Parameters

struct snd_card * card the card instance

void * device_data the data pointer to disconnect

Description

Turns the device into the disconnection state, invoking dev_disconnect callback, if the device was already registered.

Usually called from `snd_card_disconnect()`.

Return

Zero if successful, or a negative error code on failure or if the device not found.

void **snd_device_free**(struct snd_card * card, void * device_data)
release the device from the card

Parameters

struct snd_card * card the card instance

void * device_data the data pointer to release

Description

Removes the device from the list on the card and invokes the callbacks, dev_disconnect and dev_free, corresponding to the state. Then release the device.

int **snd_device_register**(struct snd_card * card, void * device_data)
register the device

Parameters

struct snd_card * card the card instance

void * device_data the data pointer to register

Description

Registers the device which was already created via `snd_device_new()`. Usually this is called from `snd_card_register()`, but it can be called later if any new devices are created after invocation of `snd_card_register()`.

Return

Zero if successful, or a negative error code on failure or if the device not found.

int **snd_device_get_state**(struct snd_card * card, void * device_data)
Get the current state of the given device

Parameters

struct snd_card * card the card instance

void * device_data the data pointer to release

Description

Returns the current state of the given device object. For the valid device, either **SNDRV_DEV_BUILD**, **SNDRV_DEV_REGISTERED** or **SNDRV_DEV_DISCONNECTED** is returned. Or for a non-existing device, -1 is returned as an error.

int **snd_info_get_line**(struct snd_info_buffer * buffer, char * line, int len)
read one line from the procfs buffer

Parameters

struct snd_info_buffer * buffer the procfs buffer

char * line the buffer to store

int len the max. buffer size

Description

Reads one line from the buffer and stores the string.

Return

Zero if successful, or 1 if error or EOF.

const char * **snd_info_get_str**(char * dest, const char * src, int len)
parse a string token

Parameters

char * dest the buffer to store the string token

const char * src the original string

int len the max. length of token - 1

Description

Parses the original string and copy a token to the given string buffer.

Return

The updated pointer of the original string so that it can be used for the next call.

struct snd_info_entry * **snd_info_create_module_entry**(struct module * module, const char * name, struct snd_info_entry * parent)
create an info entry for the given module

Parameters

struct module * module the module pointer

const char * name the file name

struct snd_info_entry * parent the parent directory

Description

Creates a new info entry and assigns it to the given module.

Return

The pointer of the new instance, or `NULL` on failure.

struct snd_info_entry * **snd_info_create_card_entry**(struct snd_card * card, const char * name, struct snd_info_entry * parent)
create an info entry for the given card

Parameters

struct snd_card * card the card instance

const char * name the file name

struct snd_info_entry * parent the parent directory

Description

Creates a new info entry and assigns it to the given card.

Return

The pointer of the new instance, or `NULL` on failure.

void **snd_info_free_entry**(struct snd_info_entry * entry)
release the info entry

Parameters

struct snd_info_entry * entry the info entry

Description

Releases the info entry.

int **snd_info_register**(struct snd_info_entry * entry)
register the info entry

Parameters

struct snd_info_entry * entry the info entry

Description

Registers the proc info entry. The all children entries are registered recursively.

Return

Zero if successful, or a negative error code on failure.

int **snd_card_rw_proc_new**(struct snd_card * card, const char * name, void * private_data, void (*read)(struct snd_info_entry *, struct snd_info_buffer *), void (*write) (struct snd_info_entry *entry, struct snd_info_buffer *buffer))

Create a read/write text proc file entry for the card

Parameters

struct snd_card * card the card instance

const char * name the file name

void * private_data the arbitrary private data

void (*)(struct snd_info_entry *, struct snd_info_buffer *) read the read callback

void (*)(struct snd_info_entry *entry, struct snd_info_buffer *buffer) write the write callback, NULL for read-only

Description

This proc file entry will be registered via `snd_card_register()` call, and it will be removed automatically at the card removal, too.

int **snd_rawmidi_receive**(struct snd_rawmidi_substream * substream, const unsigned char * buffer, int count)

receive the input data from the device

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

const unsigned char * buffer the buffer pointer

int count the data size to read

Description

Reads the data from the internal buffer.

Return

The size of read data, or a negative error code on failure.

int **snd_rawmidi_transmit_empty**(struct snd_rawmidi_substream * substream)

check whether the output buffer is empty

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

Return

1 if the internal output buffer is empty, 0 if not.

int **__snd_rawmidi_transmit_peek**(struct snd_rawmidi_substream * substream, unsigned char * buffer, int count)
copy data from the internal buffer

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

unsigned char * buffer the buffer pointer

int count data size to transfer

Description

This is a variant of `snd_rawmidi_transmit_peek()` without spinlock.

int **snd_rawmidi_transmit_peek**(struct snd_rawmidi_substream * substream, unsigned char * buffer, int count)
copy data from the internal buffer

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

unsigned char * buffer the buffer pointer

int count data size to transfer

Description

Copies data from the internal output buffer to the given buffer.

Call this in the interrupt handler when the midi output is ready, and call `snd_rawmidi_transmit_ack()` after the transmission is finished.

Return

The size of copied data, or a negative error code on failure.

int **__snd_rawmidi_transmit_ack**(struct snd_rawmidi_substream * substream, int count)
acknowledge the transmission

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

int count the transferred count

Description

This is a variant of `__snd_rawmidi_transmit_ack()` without spinlock.

int **snd_rawmidi_transmit_ack**(struct snd_rawmidi_substream * substream, int count)
acknowledge the transmission

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

int count the transferred count

Description

Advances the hardware pointer for the internal output buffer with the given size and updates the condition. Call after the transmission is finished.

Return

The advanced size if successful, or a negative error code on failure.

int **snd_rawmidi_transmit**(struct snd_rawmidi_substream * substream, unsigned char * buffer, int count)
copy from the buffer to the device

Parameters

struct snd_rawmidi_substream * substream the rawmidi substream

unsigned char * buffer the buffer pointer

int count the data size to transfer

Description

Copies data from the buffer to the device and advances the pointer.

Return

The copied size if successful, or a negative error code on failure.

int **snd_rawmidi_proceed**(struct snd_rawmidi_substream * substream)
Discard the all pending bytes and proceed

Parameters

struct snd_rawmidi_substream * substream rawmidi substream

Return

the number of discarded bytes

int **snd_rawmidi_new**(struct snd_card * card, char * id, int device, int output_count, int input_count, struct snd_rawmidi ** rrawmidi)
create a rawmidi instance

Parameters

struct snd_card * card the card instance

char * id the id string

int device the device index

int output_count the number of output streams

int input_count the number of input streams

struct snd_rawmidi ** rrawmidi the pointer to store the new rawmidi instance

Description

Creates a new rawmidi instance. Use `snd_rawmidi_set_ops()` to set the operators to the new instance.

Return

Zero if successful, or a negative error code on failure.

void **snd_rawmidi_set_ops**(struct snd_rawmidi * rmidi, int stream, const struct snd_rawmidi_ops * ops)
set the rawmidi operators

Parameters

struct snd_rawmidi * rmidi the rawmidi instance

int stream the stream direction, SNDRV_RAWMIDI_STREAM_XXX

const struct snd_rawmidi_ops * ops the operator table

Description

Sets the rawmidi operators for the given stream direction.

void **snd_request_card**(int card)
try to load the card module

Parameters

int card the card number

Description

Tries to load the module "snd-card-X" for the given card number via request_module. Returns immediately if already loaded.

void * **snd_lookup_minor_data**(unsigned int minor, int type)
get user data of a registered device

Parameters

unsigned int minor the minor number

int type device type (SNDRV_DEVICE_TYPE_XXX)

Description

Checks that a minor device with the specified type is registered, and returns its user data pointer.

This function increments the reference counter of the card instance if an associated instance with the given minor number and type is found. The caller must call `snd_card_unref()` appropriately later.

Return

The user data pointer if the specified device is found. `NULL` otherwise.

int **snd_register_device**(int type, struct snd_card * card, int dev, const struct file_operations * f_ops, void * private_data, struct device * device)
Register the ALSA device file for the card

Parameters

int type the device type, SNDRV_DEVICE_TYPE_XXX

struct snd_card * card the card instance

int dev the device index

const struct file_operations * f_ops the file operations

void * private_data user pointer for f_ops->open()

struct device * device the device to register

Description

Registers an ALSA device file for the given card. The operators have to be set in reg parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_unregister_device**(struct device * dev)
unregister the device on the given card

Parameters

struct device * dev the device instance

Description

Unregisters the device file already registered via `snd_register_device()`.

Return

Zero if successful, or a negative error code on failure.

int **copy_to_user_fromio**(void __user * dst, const volatile void __iomem * src, size_t count)
copy data from mmio-space to user-space

Parameters

void __user * dst the destination pointer on user-space

const volatile void __iomem * src the source pointer on mmio

size_t count the data size to copy in bytes

Description

Copies the data from mmio-space to user-space.

Return

Zero if successful, or non-zero on failure.

int **copy_from_user_toio**(volatile void __iomem * dst, const void __user * src, size_t count)
copy data from user-space to mmio-space

Parameters

volatile void __iomem * dst the destination pointer on mmio-space

const void __user * src the source pointer on user-space

size_t count the data size to copy in bytes

Description

Copies the data from user-space to mmio-space.

Return

Zero if successful, or non-zero on failure.

void **snd_pcm_lib_preallocate_free_for_all**(struct snd_pcm * pcm)
release all pre-allocated buffers on the pcm

Parameters

struct snd_pcm * pcm the pcm instance

Description

Releases all the pre-allocated buffers on the given pcm.

void **snd_pcm_lib_preallocate_pages**(struct snd_pcm_substream * substream, int type, struct device * data, size_t size, size_t max)
pre-allocation for the given DMA type

Parameters

struct snd_pcm_substream * substream the pcm substream instance

int type DMA type (SNDRV_DMA_TYPE_*)

struct device * data DMA type dependent data

size_t size the requested pre-allocation size in bytes

size_t max the max. allowed pre-allocation size

Description

Do pre-allocation for the given DMA buffer type.

void **snd_pcm_lib_preallocate_pages_for_all**(struct snd_pcm * pcm, int type, void * data, size_t size, size_t max)
pre-allocation for continuous memory type (all substreams)

Parameters

struct snd_pcm * pcm the pcm instance

int type DMA type (SNDRV_DMA_TYPE_*)

void * data DMA type dependent data

size_t size the requested pre-allocation size in bytes

size_t max the max. allowed pre-allocation size

Description

Do pre-allocation to all substreams of the given pcm for the specified DMA type.

void **snd_pcm_set_managed_buffer**(struct snd_pcm_substream * substream, int type, struct device * data, size_t size, size_t max)
set up buffer management for a substream

Parameters

struct snd_pcm_substream * substream the pcm substream instance

int type DMA type (SNDRV_DMA_TYPE_*)

struct device * data DMA type dependent data

size_t size the requested pre-allocation size in bytes

size_t max the max. allowed pre-allocation size

Description

Do pre-allocation for the given DMA buffer type, and set the managed buffer allocation mode to the given substream. In this mode, PCM core will allocate a buffer automatically before PCM hw_params ops call, and release the buffer after PCM hw_free ops call as well, so that the driver doesn't need to invoke the allocation and the release explicitly in its callback. When a buffer is actually allocated before the PCM hw_params call, it turns on the runtime buffer_changed flag for drivers changing their h/w parameters accordingly.

void **snd_pcm_set_managed_buffer_all**(struct snd_pcm * pcm, int type, struct device * data, size_t size, size_t max)
set up buffer management for all substreams for all substreams

Parameters

struct snd_pcm * pcm the pcm instance

int type DMA type (SNDRV_DMA_TYPE_*)

struct device * data DMA type dependent data

size_t size the requested pre-allocation size in bytes

size_t max the max. allowed pre-allocation size

Description

Do pre-allocation to all substreams of the given pcm for the specified DMA type and size, and set the managed_buffer_alloc flag to each substream.

int **snd_pcm_lib_malloc_pages**(struct snd_pcm_substream * substream, size_t size)
allocate the DMA buffer

Parameters

struct snd_pcm_substream * substream the substream to allocate the DMA buffer to

size_t size the requested buffer size in bytes

Description

Allocates the DMA buffer on the BUS type given earlier to snd_pcm_lib_preallocate_xxx_pages().

Return

1 if the buffer is changed, 0 if not changed, or a negative code on failure.

int **snd_pcm_lib_free_pages**(struct snd_pcm_substream * substream)
release the allocated DMA buffer.

Parameters

struct snd_pcm_substream * substream the substream to release the DMA buffer

Description

Releases the DMA buffer allocated via `snd_pcm_lib_malloc_pages()`.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_lib_free_vmalloc_buffer**(struct snd_pcm_substream * substream)
free vmalloc buffer

Parameters

struct snd_pcm_substream * substream the substream with a buffer allocated by `snd_pcm_lib_alloc_vmalloc_buffer()`

Return

Zero if successful, or a negative error code on failure.

struct page * **snd_pcm_lib_get_vmalloc_page**(struct snd_pcm_substream * substream, unsigned long offset)
map vmalloc buffer offset to page struct

Parameters

struct snd_pcm_substream * substream the substream with a buffer allocated by `snd_pcm_lib_alloc_vmalloc_buffer()`

unsigned long offset offset in the buffer

Description

This function is to be used as the page callback in the PCM ops.

Return

The page struct, or `NULL` on failure.

void **snd_device_initialize**(struct device * dev, struct snd_card * card)
Initialize struct device for sound devices

Parameters

struct device * dev device to initialize

struct snd_card * card card to assign, optional

int **snd_card_new**(struct device * parent, int idx, const char * xid, struct module * module, int extra_size, struct snd_card ** card_ret)
create and initialize a soundcard structure

Parameters

struct device * parent the parent device object

int idx card index (address) [0 …(SNDRV_CARDS-1)]

const char * xid card identification (ASCII string)

struct module * module top level module for locking

int extra_size allocate this extra size after the main soundcard structure

struct snd_card ** card_ret the pointer to store the created card instance

Creates and initializes a soundcard structure.

The function allocates snd_card instance via kzalloc with the given space for the driver to use freely. The allocated struct is stored in the given card_ret pointer.

Return

Zero if successful or a negative error code.

struct snd_card * **snd_card_ref**(int idx)
Get the card object from the index

Parameters

int idx the card index

Description

Returns a card object corresponding to the given index or NULL if not found. Release the object via `snd_card_unref()`.

int **snd_card_disconnect**(struct snd_card * card)
disconnect all APIs from the file-operations (user space)

Parameters

struct snd_card * card soundcard structure

Disconnects all APIs from the file-operations (user space).

Return

Zero, otherwise a negative error code.

Note

The current implementation replaces all active file->f_op with special
dummy file operations (they do nothing except release).

void **snd_card_disconnect_sync**(struct snd_card * card)
disconnect card and wait until files get closed

Parameters

struct snd_card * card card object to disconnect

Description

This calls `snd_card_disconnect()` for disconnecting all belonging components and waits until all pending files get closed. It assures that all accesses from user-space finished so that the driver can release its resources gracefully.

int **snd_card_free_when_closed**(struct snd_card * card)
Disconnect the card, free it later eventually

Parameters

struct snd_card * card soundcard structure

Description

Unlike `snd_card_free()`, this function doesn't try to release the card resource immediately, but tries to disconnect at first. When the card is still in use, the function returns before freeing the resources. The card resources will be freed when the refcount gets to zero.

int **snd_card_free**(struct snd_card * card)
frees given soundcard structure

Parameters

struct snd_card * card soundcard structure

Description

This function releases the soundcard structure and the all assigned devices automatically. That is, you don't have to release the devices by yourself.

This function waits until the all resources are properly released.

Return

Zero. Frees all associated devices and frees the control interface associated to given soundcard.

void **snd_card_set_id**(struct snd_card * card, const char * nid)
set card identification name

Parameters

struct snd_card * card soundcard structure

const char * nid new identification string

This function sets the card identification and checks for name collisions.

int **snd_card_add_dev_attr**(struct snd_card * card, const struct attribute_group * group)
Append a new sysfs attribute group to card

Parameters

struct snd_card * card card instance

const struct attribute_group * group attribute group to append

int **snd_card_register**(struct snd_card * card)
register the soundcard

Parameters

struct snd_card * card soundcard structure

This function registers all the devices assigned to the soundcard. Until calling this, the ALSA control interface is blocked from the external accesses. Thus, you should call this function at the end of the initialization of the card.

Return

Zero otherwise a negative error code if the registration failed.

int **snd_component_add**(struct snd_card * card, const char * component)
add a component string

Parameters

struct snd_card * card soundcard structure

const char * component the component id string

This function adds the component id string to the supported list. The component can be referred from the alsa-lib.

Return

Zero otherwise a negative error code.

int **snd_card_file_add**(struct snd_card * card, struct file * file)
add the file to the file list of the card

Parameters

struct snd_card * card soundcard structure

struct file * file file pointer

This function adds the file to the file linked-list of the card. This linked-list is used to keep tracking the connection state, and to avoid the release of busy resources by hotplug.

Return

zero or a negative error code.

int **snd_card_file_remove**(struct snd_card * card, struct file * file)
remove the file from the file list

Parameters

struct snd_card * card soundcard structure

struct file * file file pointer

This function removes the file formerly added to the card via `snd_card_file_add()` function. If all files are removed and `snd_card_free_when_closed()` was called beforehand, it processes the pending release of resources.

Return

Zero or a negative error code.

int **snd_power_wait**(struct snd_card * card, unsigned int power_state)
wait until the power-state is changed.

Parameters

struct snd_card * card soundcard structure

unsigned int power_state expected power state

Waits until the power-state is changed.

Return

Zero if successful, or a negative error code.

void **snd_dma_program**(unsigned long dma, unsigned long addr, unsigned int size, unsigned short mode)
program an ISA DMA transfer

Parameters

unsigned long dma the dma number

unsigned long addr the physical address of the buffer

unsigned int size the DMA transfer size

unsigned short mode the DMA transfer mode, DMA_MODE_XXX

Description

Programs an ISA DMA transfer for the given buffer.

void **snd_dma_disable**(unsigned long dma)
stop the ISA DMA transfer

Parameters

unsigned long dma the dma number

Description

Stops the ISA DMA transfer.

unsigned int **snd_dma_pointer**(unsigned long dma, unsigned int size)
return the current pointer to DMA transfer buffer in bytes

Parameters

unsigned long dma the dma number

unsigned int size the dma transfer size

Return

The current pointer in DMA transfer buffer in bytes.

void **snd_ctl_notify**(struct snd_card * card, unsigned int mask, struct snd_ctl_elem_id * id)
Send notification to user-space for a control change

Parameters

struct snd_card * card the card to send notification

unsigned int mask the event mask, SNDRV_CTL_EVENT_*

struct snd_ctl_elem_id * id the ctl element id to send notification

Description

This function adds an event record with the given id and mask, appends to the list and wakes up the user-space for notification. This can be called in the atomic context.

struct snd_kcontrol * **snd_ctl_new1**(const struct snd_kcontrol_new * ncontrol, void * private_data)
create a control instance from the template

Parameters

const struct snd_kcontrol_new * ncontrol the initialization record

void * private_data the private data to set

Description

Allocates a new struct snd_kcontrol instance and initialize from the given template. When the access field of ncontrol is 0, it's assumed as READWRITE access. When the count field is 0, it's assumes as one.

Return

The pointer of the newly generated instance, or NULL on failure.

void **snd_ctl_free_one**(struct snd_kcontrol * kcontrol)
release the control instance

Parameters

struct snd_kcontrol * kcontrol the control instance

Description

Releases the control instance created via snd_ctl_new() or snd_ctl_new1(). Don't call this after the control was added to the card.

int **snd_ctl_add**(struct snd_card * card, struct snd_kcontrol * kcontrol)
add the control instance to the card

Parameters

struct snd_card * card the card instance

struct snd_kcontrol * kcontrol the control instance to add

Description

Adds the control instance created via snd_ctl_new() or snd_ctl_new1() to the given card. Assigns also an unique numid used for fast search.

It frees automatically the control which cannot be added.

Return

Zero if successful, or a negative error code on failure.

int **snd_ctl_replace**(struct snd_card * card, struct snd_kcontrol * kcontrol,
bool add_on_replace)
replace the control instance of the card

Parameters

struct snd_card * card the card instance

struct snd_kcontrol * kcontrol the control instance to replace

bool add_on_replace add the control if not already added

Description

Replaces the given control. If the given control does not exist and the add_on_replace flag is set, the control is added. If the control exists, it is destroyed first.

It frees automatically the control which cannot be added or replaced.

Return

Zero if successful, or a negative error code on failure.

int **snd_ctl_remove**(struct snd_card * card, struct snd_kcontrol * kcontrol)
remove the control from the card and release it

Parameters

struct snd_card * card the card instance

struct snd_kcontrol * kcontrol the control instance to remove

Description

Removes the control from the card and then releases the instance. You don't need to call `snd_ctl_free_one()`. You must be in the write lock - down_write(`card->controls_rwsem`).

Return

0 if successful, or a negative error code on failure.

int **snd_ctl_remove_id**(struct snd_card * card, struct snd_ctl_elem_id * id)
remove the control of the given id and release it

Parameters

struct snd_card * card the card instance

struct snd_ctl_elem_id * id the control id to remove

Description

Finds the control instance with the given id, removes it from the card list and releases it.

Return

0 if successful, or a negative error code on failure.

int **snd_ctl_activate_id**(struct snd_card * card, struct snd_ctl_elem_id * id, int active)
activate/inactivate the control of the given id

Parameters

struct snd_card * card the card instance

struct snd_ctl_elem_id * id the control id to activate/inactivate

int active non-zero to activate

Description

Finds the control instance with the given id, and activate or inactivate the control together with notification, if changed. The given ID data is filled with full information.

Return

0 if unchanged, 1 if changed, or a negative error code on failure.

int **snd_ctl_rename_id**(struct snd_card * card, struct snd_ctl_elem_id * src_id, struct snd_ctl_elem_id * dst_id)
replace the id of a control on the card

Parameters

struct snd_card * card the card instance

struct snd_ctl_elem_id * src_id the old id

struct snd_ctl_elem_id * dst_id the new id

Description

Finds the control with the old id from the card, and replaces the id with the new one.

Return

Zero if successful, or a negative error code on failure.

struct snd_kcontrol * **snd_ctl_find_numid**(struct snd_card * card, unsigned int numid)
find the control instance with the given number-id

Parameters

struct snd_card * card the card instance

unsigned int numid the number-id to search

Description

Finds the control instance with the given number-id from the card.

The caller must down card->controls_rwsem before calling this function (if the race condition can happen).

Return

The pointer of the instance if found, or `NULL` if not.

struct snd_kcontrol * **snd_ctl_find_id**(struct snd_card * card, struct snd_ctl_elem_id * id)
find the control instance with the given id

Parameters

struct snd_card * card the card instance

struct snd_ctl_elem_id * id the id to search

Description

Finds the control instance with the given id from the card.

The caller must down card->controls_rwsem before calling this function (if the race condition can happen).

Return

The pointer of the instance if found, or `NULL` if not.

int **snd_ctl_register_ioctl**(snd_kctl_ioctl_func_t fcn)
register the device-specific control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn ioctl callback function

Description

called from each device manager like pcm.c, hwdep.c, etc.

int **snd_ctl_register_ioctl_compat**(snd_kctl_ioctl_func_t fcn)
register the device-specific 32bit compat control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn ioctl callback function

int **snd_ctl_unregister_ioctl**(snd_kctl_ioctl_func_t fcn)
de-register the device-specific control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn ioctl callback function to unregister

int **snd_ctl_unregister_ioctl_compat**(snd_kctl_ioctl_func_t fcn)
de-register the device-specific compat 32bit control-ioctls

Parameters

snd_kctl_ioctl_func_t fcn ioctl callback function to unregister

int **snd_ctl_boolean_mono_info**(struct snd_kcontrol * kcontrol, struct snd_ctl_elem_info * uinfo)
Helper function for a standard boolean info callback with a mono channel

Parameters

struct snd_kcontrol * kcontrol the kcontrol instance

struct snd_ctl_elem_info * uinfo info to store

Description

This is a function that can be used as info callback for a standard boolean control with a single mono channel.

int **snd_ctl_boolean_stereo_info**(struct snd_kcontrol * kcontrol, struct snd_ctl_elem_info * uinfo)
Helper function for a standard boolean info callback with stereo two channels

Parameters

struct snd_kcontrol * kcontrol the kcontrol instance

struct snd_ctl_elem_info * uinfo info to store

Description

This is a function that can be used as info callback for a standard boolean control with stereo two channels.

int **snd_ctl_enum_info**(struct snd_ctl_elem_info * info, unsigned int channels, unsigned int items, const char *const names)
fills the info structure for an enumerated control

Parameters

struct snd_ctl_elem_info * info the structure to be filled

unsigned int channels the number of the control's channels; often one

unsigned int items the number of control values; also the size of **names**

const char *const names an array containing the names of all control values

Description

Sets all required fields in **info** to their appropriate values. If the control's accessibility is not the default (readable and writable), the caller has to fill **info->access**.

Return

Zero.

void **snd_pcm_set_ops**(struct snd_pcm * pcm, int direction, const struct snd_pcm_ops * ops)
set the PCM operators

Parameters

struct snd_pcm * pcm the pcm instance

int direction stream direction, SNDRV_PCM_STREAM_XXX

const struct snd_pcm_ops * ops the operator table

Description

Sets the given PCM operators to the pcm instance.

void **snd_pcm_set_sync**(struct snd_pcm_substream * substream)
set the PCM sync id

Parameters

struct snd_pcm_substream * substream the pcm substream

Description

Sets the PCM sync identifier for the card.

int **snd_interval_refine**(struct snd_interval * i, const struct snd_interval * v)
refine the interval value of configurator

Parameters

struct snd_interval * i the interval value to refine

const struct snd_interval * v the interval value to refer to

Description

Refines the interval value with the reference value. The interval is changed to the range satisfying both intervals. The interval status (min, max, integer, etc.) are evaluated.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_interval_ratnum**(struct snd_interval * i, unsigned int rats_count, const struct snd_ratnum * rats, unsigned int * nump, unsigned int * denp)
refine the interval value

Parameters

struct snd_interval * i interval to refine

unsigned int rats_count number of ratnum_t

const struct snd_ratnum * rats ratnum_t array

unsigned int * nump pointer to store the resultant numerator

unsigned int * denp pointer to store the resultant denominator

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_interval_list**(struct snd_interval * i, unsigned int count, const unsigned int * list, unsigned int mask)
refine the interval value from the list

Parameters

struct snd_interval * i the interval value to refine

unsigned int count the number of elements in the list

const unsigned int * list the value list

unsigned int mask the bit-mask to evaluate

Description

Refines the interval value from the list. When mask is non-zero, only the elements corresponding to bit 1 are evaluated.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_interval_ranges**(struct snd_interval * i, unsigned int count, const struct snd_interval * ranges, unsigned int mask)
refine the interval value from the list of ranges

Parameters

struct snd_interval * i the interval value to refine

unsigned int count the number of elements in the list of ranges

const struct snd_interval * ranges the ranges list

unsigned int mask the bit-mask to evaluate

Description

Refines the interval value from the list of ranges. When mask is non-zero, only the elements corresponding to bit 1 are evaluated.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_hw_rule_add**(struct snd_pcm_runtime * runtime, unsigned int cond, int var, snd_pcm_hw_rule_func_t func, void * private, int dep, ...)

add the hw-constraint rule

Parameters

struct snd_pcm_runtime * runtime the pcm runtime instance

unsigned int cond condition bits

int var the variable to evaluate

snd_pcm_hw_rule_func_t func the evaluation function

void * private the private data pointer passed to function

int dep the dependent variables

... variable arguments

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_mask64**(struct snd_pcm_runtime * runtime, snd_pcm_hw_param_t var, u_int64_t mask)

apply the given bitmap mask constraint

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_hw_param_t var hw_params variable to apply the mask

u_int64_t mask the 64bit bitmap mask

Description

Apply the constraint of the given bitmap mask to a 64-bit mask parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_integer**(struct snd_pcm_runtime * runtime, snd_pcm_hw_param_t var)

apply an integer constraint to an interval

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_hw_param_t var hw_params variable to apply the integer constraint

Description

Apply the constraint of integer to an interval parameter.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_hw_constraint_minmax**(struct snd_pcm_runtime * runtime, snd_pcm_hw_param_t var, unsigned int min, unsigned int max)

apply a min/max range constraint to an interval

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

snd_pcm_hw_param_t var hw_params variable to apply the range

unsigned int min the minimal value

unsigned int max the maximal value

Description

Apply the min/max range constraint to an interval parameter.

Return

Positive if the value is changed, zero if it's not changed, or a negative error code.

int **snd_pcm_hw_constraint_list**(struct snd_pcm_runtime * runtime, unsigned int cond, snd_pcm_hw_param_t var, const struct snd_pcm_hw_constraint_list * l)

apply a list of constraints to a parameter

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the list constraint

const struct snd_pcm_hw_constraint_list * l list

Description

Apply the list of constraints to an interval parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_ranges**(struct snd_pcm_runtime * runtime, unsigned int cond, snd_pcm_hw_param_t var, const struct snd_pcm_hw_constraint_ranges * r)

apply list of range constraints to a parameter

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the list of range constraints

const struct snd_pcm_hw_constraint_ranges * r ranges

Description

Apply the list of range constraints to an interval parameter.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_ratnums**(struct snd_pcm_runtime * runtime, unsigned int cond, snd_pcm_hw_param_t var, const struct snd_pcm_hw_constraint_ratnums * r)

apply ratnums constraint to a parameter

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the ratnums constraint

const struct snd_pcm_hw_constraint_ratnums * r struct snd_ratnums constriants

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_ratdens**(struct snd_pcm_runtime * runtime, unsigned int cond, snd_pcm_hw_param_t var, const struct snd_pcm_hw_constraint_ratdens * r)

apply ratdens constraint to a parameter

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the ratdens constraint

const struct snd_pcm_hw_constraint_ratdens * r struct snd_ratdens constriants

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_msbits**(struct snd_pcm_runtime * runtime, unsigned int cond, unsigned int width, unsigned int msbits)

add a hw constraint msbits rule

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

unsigned int width sample bits width

unsigned int msbits msbits width

Description

This constraint will set the number of most significant bits (msbits) if a sample format with the specified width has been select. If width is set to 0 the msbits will be set for any sample format with a width larger than the specified msbits.

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_step**(struct snd_pcm_runtime * runtime, unsigned int cond, snd_pcm_hw_param_t var, unsigned long step)

add a hw constraint step rule

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the step constraint

unsigned long step step size

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_constraint_pow2**(struct snd_pcm_runtime * runtime, unsigned int cond, snd_pcm_hw_param_t var)

add a hw constraint power-of-2 rule

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int cond condition bits

snd_pcm_hw_param_t var hw_params variable to apply the power-of-2 constraint

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_rule_noresample**(struct snd_pcm_runtime * runtime, unsigned int base_rate)

add a rule to allow disabling hw resampling

Parameters

struct snd_pcm_runtime * runtime PCM runtime instance

unsigned int base_rate the rate at which the hardware does not resample

Return

Zero if successful, or a negative error code on failure.

int **snd_pcm_hw_param_value**(const struct snd_pcm_hw_params * params, snd_pcm_hw_param_t var, int * dir)
return **params** field **var** value

Parameters

const struct snd_pcm_hw_params * params the hw_params instance

snd_pcm_hw_param_t var parameter to retrieve

int * dir pointer to the direction (-1,0,1) or NULL

Return

The value for field **var** if it's fixed in configuration space defined by **params**. -EINVAL otherwise.

int **snd_pcm_hw_param_first**(struct snd_pcm_substream * pcm, struct snd_pcm_hw_params * params, snd_pcm_hw_param_t var, int * dir)
refine config space and return minimum value

Parameters

struct snd_pcm_substream * pcm PCM instance

struct snd_pcm_hw_params * params the hw_params instance

snd_pcm_hw_param_t var parameter to retrieve

int * dir pointer to the direction (-1,0,1) or NULL

Description

Inside configuration space defined by **params** remove from **var** all values > minimum. Reduce configuration space accordingly.

Return

The minimum, or a negative error code on failure.

int **snd_pcm_hw_param_last**(struct snd_pcm_substream * pcm, struct snd_pcm_hw_params * params, snd_pcm_hw_param_t var, int * dir)
refine config space and return maximum value

Parameters

struct snd_pcm_substream * pcm PCM instance

struct snd_pcm_hw_params * params the hw_params instance

snd_pcm_hw_param_t var parameter to retrieve

int * dir pointer to the direction (-1,0,1) or NULL

Description

Inside configuration space defined by **params** remove from **var** all values < maximum. Reduce configuration space accordingly.

Return

The maximum, or a negative error code on failure.

int **snd_pcm_lib_ioctl**(struct snd_pcm_substream * substream, unsigned int cmd, void * arg)
a generic PCM ioctl callback

Parameters

struct snd_pcm_substream * substream the pcm substream instance

unsigned int cmd ioctl command

void * arg ioctl argument

Description

Processes the generic ioctl commands for PCM. Can be passed as the ioctl callback for PCM ops.

Return

Zero if successful, or a negative error code on failure.

void **snd_pcm_period_elapsed**(struct snd_pcm_substream * substream)
update the pcm status for the next period

Parameters

struct snd_pcm_substream * substream the pcm substream instance

Description

This function is called from the interrupt handler when the PCM has processed the period size. It will update the current pointer, wake up sleepers, etc.

Even if more than one periods have elapsed since the last call, you have to call this only once.

int **snd_pcm_add_chmap_ctls**(struct snd_pcm * pcm, int stream, const struct snd_pcm_chmap_elem * chmap, int max_channels, unsigned long private_value, struct snd_pcm_chmap ** info_ret)
create channel-mapping control elements

Parameters

struct snd_pcm * pcm the assigned PCM instance

int stream stream direction

const struct snd_pcm_chmap_elem * chmap channel map elements (for query)

int max_channels the max number of channels for the stream

unsigned long private_value the value passed to each kcontrol's private_value field

struct snd_pcm_chmap ** info_ret store struct snd_pcm_chmap instance if non-NULL

Description

Create channel-mapping control elements assigned to the given PCM stream(s).

Return

Zero if successful, or a negative error value.

int **snd_hwdep_new**(struct snd_card * card, char * id, int device, struct snd_hwdep ** rhwdep)
create a new hwdep instance

Parameters

struct snd_card * card the card instance

char * id the id string

int device the device index (zero-based)

struct snd_hwdep ** rhwdep the pointer to store the new hwdep instance

Description

Creates a new hwdep instance with the given index on the card. The callbacks (hwdep->ops) must be set on the returned instance after this call manually by the caller.

Return

Zero if successful, or a negative error code on failure.

void **snd_pcm_stream_lock**(struct snd_pcm_substream * substream)
Lock the PCM stream

Parameters

struct snd_pcm_substream * substream PCM substream

Description

This locks the PCM stream's spinlock or mutex depending on the nonatomic flag of the given substream. This also takes the global link rw lock (or rw sem), too, for avoiding the race with linked streams.

void **snd_pcm_stream_unlock**(struct snd_pcm_substream * substream)
Unlock the PCM stream

Parameters

struct snd_pcm_substream * substream PCM substream

Description

This unlocks the PCM stream that has been locked via `snd_pcm_stream_lock()`.

void **snd_pcm_stream_lock_irq**(struct snd_pcm_substream * substream)
Lock the PCM stream

Parameters

struct snd_pcm_substream * substream PCM substream

Description

This locks the PCM stream like `snd_pcm_stream_lock()` and disables the local IRQ (only when nonatomic is false). In nonatomic case, this is identical as `snd_pcm_stream_lock()`.

void **snd_pcm_stream_unlock_irq**(struct snd_pcm_substream * substream)
Unlock the PCM stream

Parameters

struct snd_pcm_substream * substream PCM substream

Description

This is a counter-part of `snd_pcm_stream_lock_irq()`.

void **snd_pcm_stream_unlock_irqrestore**(struct snd_pcm_substream * substream, unsigned long flags)
Unlock the PCM stream

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned long flags irq flags

Description

This is a counter-part of `snd_pcm_stream_lock_irqsave()`.

int **snd_pcm_stop**(struct snd_pcm_substream * substream, snd_pcm_state_t state)
try to stop all running streams in the substream group

Parameters

struct snd_pcm_substream * substream the PCM substream instance

snd_pcm_state_t state PCM state after stopping the stream

Description

The state of each stream is then changed to the given state unconditionally.

Return

Zero if successful, or a negative error code.

int **snd_pcm_stop_xrun**(struct snd_pcm_substream * substream)
stop the running streams as XRUN

Parameters

struct snd_pcm_substream * substream the PCM substream instance

Description

This stops the given running substream (and all linked substreams) as XRUN. Unlike `snd_pcm_stop()`, this function takes the substream lock by itself.

Return

Zero if successful, or a negative error code.

int **snd_pcm_suspend_all**(struct snd_pcm * pcm)
trigger SUSPEND to all substreams in the given pcm

Parameters

struct snd_pcm * pcm the PCM instance

Description

After this call, all streams are changed to SUSPENDED state.

Return

Zero if successful (or **pcm** is NULL), or a negative error code.

int **snd_pcm_kernel_ioctl**(struct snd_pcm_substream * substream, unsigned int cmd, void * arg)
Execute PCM ioctl in the kernel-space

Parameters

struct snd_pcm_substream * substream PCM substream

unsigned int cmd IOCTL cmd

void * arg IOCTL argument

Description

The function is provided primarily for OSS layer and USB gadget drivers, and it allows only the limited set of ioctls (hw_params, sw_params, prepare, start, drain, drop, forward).

int **snd_pcm_lib_default_mmap**(struct snd_pcm_substream * substream, struct vm_area_struct * area)
Default PCM data mmap function

Parameters

struct snd_pcm_substream * substream PCM substream

struct vm_area_struct * area VMA

Description

This is the default mmap handler for PCM data. When mmap pcm_ops is NULL, this function is invoked implicitly.

int **snd_pcm_lib_mmap_iomem**(struct snd_pcm_substream * substream, struct vm_area_struct * area)
Default PCM data mmap function for I/O mem

Parameters

struct snd_pcm_substream * substream PCM substream

struct vm_area_struct * area VMA

Description

When your hardware uses the iomapped pages as the hardware buffer and wants to mmap it, pass this function as mmap pcm_ops. Note that this is supposed to work only on limited architectures.

int **snd_dma_alloc_pages**(int type, struct device * device, size_t size, struct snd_dma_buffer * dmab)
allocate the buffer area according to the given type

Parameters

int type the DMA buffer type

struct device * device the device pointer

size_t size the buffer size to allocate

struct snd_dma_buffer * dmab buffer allocation record to store the allocated data

Description

Calls the memory-allocator function for the corresponding buffer type.

Return

Zero if the buffer with the given size is allocated successfully, otherwise a negative value on error.

int **snd_dma_alloc_pages_fallback**(int type, struct device * device, size_t size, struct snd_dma_buffer * dmab)
allocate the buffer area according to the given type with fallback

Parameters

int type the DMA buffer type

struct device * device the device pointer

size_t size the buffer size to allocate

struct snd_dma_buffer * dmab buffer allocation record to store the allocated data

Description

Calls the memory-allocator function for the corresponding buffer type. When no space is left, this function reduces the size and tries to allocate again. The size actually allocated is stored in res_size argument.

Return

Zero if the buffer with the given size is allocated successfully, otherwise a negative value on error.

void **snd_dma_free_pages**(struct snd_dma_buffer * dmab)
release the allocated buffer

Parameters

struct snd_dma_buffer * dmab the buffer allocation record to release

Description

Releases the allocated buffer via `snd_dma_alloc_pages()`.

CHAPTER

SIXTEEN

FRAME BUFFER LIBRARY

The frame buffer drivers depend heavily on four data structures. These structures are declared in include/linux/fb.h. They are fb_info, fb_var_screeninfo, fb_fix_screeninfo and fb_monospecs. The last three can be made available to and from userland.

fb_info defines the current state of a particular video card. Inside fb_info, there exists a fb_ops structure which is a collection of needed functions to make fbdev and fbcon work. fb_info is only visible to the kernel.

fb_var_screeninfo is used to describe the features of a video card that are user defined. With fb_var_screeninfo, things such as depth and the resolution may be defined.

The next structure is fb_fix_screeninfo. This defines the properties of a card that are created when a mode is set and can't be changed otherwise. A good example of this is the start of the frame buffer memory. This "locks" the address of the frame buffer memory, so that it cannot be changed or moved.

The last structure is fb_monospecs. In the old API, there was little importance for fb_monospecs. This allowed for forbidden things such as setting a mode of 800x600 on a fix frequency monitor. With the new API, fb_monospecs prevents such things, and if used correctly, can prevent a monitor from being cooked. fb_monospecs will not be useful until kernels 2.5.x.

16.1 Frame Buffer Memory

int **remove_conflicting_framebuffers**(struct apertures_struct * a, const char * name, bool primary)
remove firmware-configured framebuffers

Parameters

struct apertures_struct * a memory range, users of which are to be removed

const char * name requesting driver name

bool primary also kick vga16fb if present

Description

This function removes framebuffer devices (initialized by firmware/bootloader) which use memory range described by **a**. If **a** is NULL all such devices are removed.

int **remove_conflicting_pci_framebuffers**(struct pci_dev * pdev, const char * name)
remove firmware-configured framebuffers for PCI devices

Parameters

struct pci_dev * pdev PCI device

const char * name requesting driver name

Description

This function removes framebuffer devices (eg. initialized by firmware) using memory range configured for any of **pdev**'s memory bars.

The function assumes that PCI device with shadowed ROM drives a primary display and so kicks out vga16fb.

int **register_framebuffer**(struct fb_info * fb_info)
registers a frame buffer device

Parameters

struct fb_info * fb_info frame buffer info structure

Registers a frame buffer device **fb_info**.

Returns negative errno on error, or zero for success.

void **unregister_framebuffer**(struct fb_info * fb_info)
releases a frame buffer device

Parameters

struct fb_info * fb_info frame buffer info structure

Unregisters a frame buffer device **fb_info**.

Returns negative errno on error, or zero for success.

This function will also notify the framebuffer console to release the driver.

This is meant to be called within a driver's `module_exit()` function. If this is called outside `module_exit()`, ensure that the driver implements fb_open() and fb_release() to check that no processes are using the device.

void **fb_set_suspend**(struct fb_info * info, int state)
low level driver signals suspend

Parameters

struct fb_info * info framebuffer affected

int state 0 = resuming, !=0 = suspending

This is meant to be used by low level drivers to signal suspend/resume to the core & clients. It must be called with the console semaphore held

16.2 Frame Buffer Colormap

void **fb_dealloc_cmap**(struct fb_cmap * cmap)
deallocate a colormap

Parameters

struct fb_cmap * cmap frame buffer colormap structure

Deallocates a colormap that was previously allocated with fb_alloc_cmap().

int **fb_copy_cmap**(const struct fb_cmap * from, struct fb_cmap * to)
copy a colormap

Parameters

const struct fb_cmap * from frame buffer colormap structure

struct fb_cmap * to frame buffer colormap structure

Copy contents of colormap from **from** to **to**.

int **fb_set_cmap**(struct fb_cmap * cmap, struct fb_info * info)
set the colormap

Parameters

struct fb_cmap * cmap frame buffer colormap structure

struct fb_info * info frame buffer info structure

Sets the colormap **cmap** for a screen of device **info**.

Returns negative errno on error, or zero on success.

const struct fb_cmap * **fb_default_cmap**(int len)
get default colormap

Parameters

int len size of palette for a depth

Gets the default colormap for a specific screen depth. **len** is the size of the palette for a particular screen depth.

Returns pointer to a frame buffer colormap structure.

void **fb_invert_cmaps**(void)
invert all defaults colormaps

Parameters

void no arguments

Description

Invert all default colormaps.

16.3 Frame Buffer Video Mode Database

int **fb_try_mode**(struct fb_var_screeninfo * var, struct fb_info * info, const struct fb_videomode * mode, unsigned int bpp)
test a video mode

Parameters

struct fb_var_screeninfo * var frame buffer user defined part of display

struct fb_info * info frame buffer info structure

const struct fb_videomode * mode frame buffer video mode structure

unsigned int bpp color depth in bits per pixel

Tries a video mode to test it's validity for device **info**.

Returns 1 on success.

void **fb_delete_videomode**(const struct fb_videomode * mode, struct list_head * head)
removed videomode entry from modelist

Parameters

const struct fb_videomode * mode videomode to remove

struct list_head * head struct list_head of modelist

NOTES

Will remove all matching mode entries

int **fb_find_mode**(struct fb_var_screeninfo * var, struct fb_info * info, const char * mode_option, const struct fb_videomode * db, unsigned int dbsize, const struct fb_videomode * default_mode, unsigned int default_bpp)
finds a valid video mode

Parameters

struct fb_var_screeninfo * var frame buffer user defined part of display

struct fb_info * info frame buffer info structure

const char * mode_option string video mode to find

const struct fb_videomode * db video mode database

unsigned int dbsize size of **db**

const struct fb_videomode * default_mode default video mode to fall back to

unsigned int default_bpp default color depth in bits per pixel

Description

Finds a suitable video mode, starting with the specified mode in **mode_option** with fallback to **default_mode**. If **default_mode** fails, all modes in the video mode database will be tried.

Valid mode specifiers for **mode_option**:

```
<xres>x<yres>[M][R][-<bpp>][@<refresh>][i][p][m]
```

or

```
<name>[-<bpp>][@<refresh>]
```

with <xres>, <yres>, <bpp> and <refresh> decimal numbers and <name> a string.

If 'M' is present after yres (and before refresh/bpp if present), the function will compute the timings using VESA(tm) Coordinated Video Timings (CVT). If 'R' is present after 'M', will compute with reduced blanking (for flatpanels). If 'i' or 'p' are present, compute interlaced or progressive mode. If 'm' is present, add margins equal to 1.8% of xres rounded down to 8 pixels, and 1.8% of yres. The char 'i', 'p' and 'm' must be after 'M' and 'R'. Example:

```
1024x768MR-8@60m - Reduced blank with margins at 60Hz.
```

Returns zero for failure, 1 if using specified **mode_option**, 2 if using specified **mode_option** with an ignored refresh rate, 3 if default mode is used, 4 if fall back to any valid mode.

NOTE

The passed struct **var** is _not_ cleared! This allows you to supply values for e.g. the grayscale and accel_flags fields.

void **fb_var_to_videomode**(struct fb_videomode * mode, const struct fb_var_screeninfo * var)
convert fb_var_screeninfo to fb_videomode

Parameters

struct fb_videomode * mode pointer to struct fb_videomode

const struct fb_var_screeninfo * var pointer to struct fb_var_screeninfo

void **fb_videomode_to_var**(struct fb_var_screeninfo * var, const struct fb_videomode * mode)
convert fb_videomode to fb_var_screeninfo

Parameters

struct fb_var_screeninfo * var pointer to struct fb_var_screeninfo

const struct fb_videomode * mode pointer to struct fb_videomode

int **fb_mode_is_equal**(const struct fb_videomode * mode1, const struct fb_videomode * mode2)
compare 2 videomodes

Parameters

const struct fb_videomode * mode1 first videomode

const struct fb_videomode * mode2 second videomode

Return

1 if equal, 0 if not

const struct fb_videomode * **fb_find_best_mode**(const struct fb_var_screeninfo * var, struct list_head * head)

find best matching videomode

Parameters

const struct fb_var_screeninfo * var pointer to struct fb_var_screeninfo

struct list_head * head pointer to struct list_head of modelist

Return

struct fb_videomode, NULL if none found

Description

IMPORTANT: This function assumes that all modelist entries in info->modelist are valid.

NOTES

Finds best matching videomode which has an equal or greater dimension than var->xres and var->yres. If more than 1 videomode is found, will return the videomode with the highest refresh rate

const struct fb_videomode * **fb_find_nearest_mode**(const struct fb_videomode * mode, struct list_head * head)

find closest videomode

Parameters

const struct fb_videomode * mode pointer to struct fb_videomode

struct list_head * head pointer to modelist

Description

Finds best matching videomode, smaller or greater in dimension. If more than 1 videomode is found, will return the videomode with the closest refresh rate.

const struct fb_videomode * **fb_match_mode**(const struct fb_var_screeninfo * var, struct list_head * head)

find a videomode which exactly matches the timings in var

Parameters

const struct fb_var_screeninfo * var pointer to struct fb_var_screeninfo

struct list_head * head pointer to struct list_head of modelist

Return

struct fb_videomode, NULL if none found

int **fb_add_videomode**(const struct fb_videomode * mode, struct list_head * head)

adds videomode entry to modelist

Parameters

const struct fb_videomode * mode videomode to add

struct list_head * head struct list_head of modelist

NOTES

Will only add unmatched mode entries

void **fb_destroy_modelist**(struct list_head * head)
destroy modelist

Parameters

struct list_head * head struct list_head of modelist

void **fb_videomode_to_modelist**(const struct fb_videomode * modedb, int num, struct list_head * head)
convert mode array to mode list

Parameters

const struct fb_videomode * modedb array of struct fb_videomode

int num number of entries in array

struct list_head * head struct list_head of modelist

16.4 Frame Buffer Macintosh Video Mode Database

int **mac_vmode_to_var**(int vmode, int cmode, struct fb_var_screeninfo * var)
converts vmode/cmode pair to var structure

Parameters

int vmode MacOS video mode

int cmode MacOS color mode

struct fb_var_screeninfo * var frame buffer video mode structure

Converts a MacOS vmode/cmode pair to a frame buffer video mode structure.

Returns negative errno on error, or zero for success.

int **mac_map_monitor_sense**(int sense)
Convert monitor sense to vmode

Parameters

int sense Macintosh monitor sense number

Converts a Macintosh monitor sense number to a MacOS vmode number.

Returns MacOS vmode video mode number.

int **mac_find_mode**(struct fb_var_screeninfo * var, struct fb_info * info, const char * mode_option, unsigned int default_bpp)
find a video mode

Parameters

struct fb_var_screeninfo * var frame buffer user defined part of display

struct fb_info * info frame buffer info structure

const char * mode_option video mode name (see mac_modedb[])

unsigned int default_bpp default color depth in bits per pixel

Finds a suitable video mode. Tries to set mode specified by **mode_option**. If the name of the wanted mode begins with ‘mac’, the Mac video mode database will be used, otherwise it will fall back to the standard video mode database.

Note

Function marked as __init and can only be used during system boot.

Returns error code from fb_find_mode (see fb_find_mode function).

16.5 Frame Buffer Fonts

Refer to the file lib/fonts/fonts.c for more information.

CHAPTER

SEVENTEEN

VOLTAGE AND CURRENT REGULATOR API

Author Liam Girdwood

Author Mark Brown

17.1 Introduction

This framework is designed to provide a standard kernel interface to control voltage and current regulators.

The intention is to allow systems to dynamically control regulator power output in order to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current limit is controllable).

Note that additional (and currently more complete) documentation is available in the Linux kernel source under `Documentation/power/regulator`.

17.1.1 Glossary

The regulator API uses a number of terms which may not be familiar:

Regulator

> Electronic device that supplies power to other devices. Most regulators can enable and disable their output and some can also control their output voltage or current.

Consumer

> Electronic device which consumes power provided by a regulator. These may either be static, requiring only a fixed supply, or dynamic, requiring active management of the regulator at runtime.

Power Domain

> The electronic circuit supplied by a given regulator, including the regulator and all consumer devices. The configuration of the regulator is shared between all the components in the circuit.

Power Management Integrated Circuit (PMIC)

An IC which contains numerous regulators and often also other subsystems. In an embedded system the primary PMIC is often equivalent to a combination of the PSU and southbridge in a desktop system.

17.2 Consumer driver interface

This offers a similar API to the kernel clock framework. Consumer drivers use get and put operations to acquire and release regulators. Functions are provided to enable and disable the regulator and to get and set the runtime parameters of the regulator.

When requesting regulators consumers use symbolic names for their supplies, such as "Vcc", which are mapped into actual regulator devices by the machine interface.

A stub version of this API is provided when the regulator framework is not in use in order to minimise the need to use ifdefs.

17.2.1 Enabling and disabling

The regulator API provides reference counted enabling and disabling of regulators. Consumer devices use the `regulator_enable()` and `regulator_disable()` functions to enable and disable regulators. Calls to the two functions must be balanced.

Note that since multiple consumers may be using a regulator and machine constraints may not allow the regulator to be disabled there is no guarantee that calling `regulator_disable()` will actually cause the supply provided by the regulator to be disabled. Consumer drivers should assume that the regulator may be enabled at all times.

17.2.2 Configuration

Some consumer devices may need to be able to dynamically configure their supplies. For example, MMC drivers may need to select the correct operating voltage for their cards. This may be done while the regulator is enabled or disabled.

The `regulator_set_voltage()` and `regulator_set_current_limit()` functions provide the primary interface for this. Both take ranges of voltages and currents, supporting drivers that do not require a specific value (eg, CPU frequency scaling normally permits the CPU to use a wider range of supply voltages at lower frequencies but does not require that the supply voltage be lowered). Where an exact value is required both minimum and maximum values should be identical.

17.2.3 Callbacks

Callbacks may also be registered for events such as regulation failures.

17.3 Regulator driver interface

Drivers for regulator chips register the regulators with the regulator core, providing operations structures to the core. A notifier interface allows error conditions to be reported to the core.

Registration should be triggered by explicit setup done by the platform, supplying a struct `regulator_init_data` for the regulator containing constraint and supply information.

17.4 Machine interface

This interface provides a way to define how regulators are connected to consumers on a given system and what the valid operating parameters are for the system.

17.4.1 Supplies

Regulator supplies are specified using struct `regulator_consumer_supply`. This is done at driver registration time as part of the machine constraints.

17.4.2 Constraints

As well as defining the connections the machine interface also provides constraints defining the operations that clients are allowed to perform and the parameters that may be set. This is required since generally regulator devices will offer more flexibility than it is safe to use on a given system, for example supporting higher supply voltages than the consumers are rated for.

This is done at driver registration time` by providing a struct `regulation_constraints`.

The constraints may also specify an initial configuration for the regulator in the constraints, which is particularly useful for use with static consumers.

17.5 API reference

Due to limitations of the kernel documentation framework and the existing layout of the source code the entire regulator API is documented here.

struct **pre_voltage_change_data**
Data sent with PRE_VOLTAGE_CHANGE event

Definition

```
struct pre_voltage_change_data {
  unsigned long old_uV;
  unsigned long min_uV;
  unsigned long max_uV;
};
```

Members

old_uV Current voltage before change.

min_uV Min voltage we'll change to.

max_uV Max voltage we'll change to.

struct **regulator_bulk_data**
Data used for bulk regulator operations.

Definition

```
struct regulator_bulk_data {
  const char *supply;
  struct regulator *consumer;
};
```

Members

supply The name of the supply. Initialised by the user before using the bulk regulator APIs.

consumer The regulator consumer for the supply. This will be managed by the bulk API.

Description

The regulator APIs provide a series of regulator_bulk_() API calls as a convenience to consumers which require multiple supplies. This structure is used to manage data for these calls.

struct **regulator_state**
regulator state during low power system states

Definition

```
struct regulator_state {
  int uV;
  int min_uV;
  int max_uV;
  unsigned int mode;
  int enabled;
  bool changeable;
};
```

Members

uV Default operating voltage during suspend, it can be adjusted among <min_uV, max_uV>.

min_uV Minimum suspend voltage may be set.

max_uV Maximum suspend voltage may be set.

mode Operating mode during suspend.

enabled operations during suspend. - DO_NOTHING_IN_SUSPEND - DISABLE_IN_SUSPEND - ENABLE_IN_SUSPEND

changeable Is this state can be switched between enabled/disabled,

Description

This describes a regulators state during a system wide low power state. One of enabled or disabled must be set for the configuration to be applied.

struct **regulation_constraints**
regulator operating constraints.

Definition

```
struct regulation_constraints {
  const char *name;
  int min_uV;
  int max_uV;
  int uV_offset;
  int min_uA;
  int max_uA;
  int ilim_uA;
  int system_load;
  u32 *max_spread;
  int max_uV_step;
  unsigned int valid_modes_mask;
  unsigned int valid_ops_mask;
  int input_uV;
  struct regulator_state state_disk;
  struct regulator_state state_mem;
  struct regulator_state state_standby;
  suspend_state_t initial_state;
  unsigned int initial_mode;
  unsigned int ramp_delay;
  unsigned int settling_time;
  unsigned int settling_time_up;
  unsigned int settling_time_down;
  unsigned int enable_time;
  unsigned int active_discharge;
  unsigned always_on:1;
  unsigned boot_on:1;
  unsigned apply_uV:1;
  unsigned ramp_disable:1;
  unsigned soft_start:1;
  unsigned pull_down:1;
  unsigned over_current_protection:1;
};
```

Members

name Descriptive name for the constraints, used for display purposes.

min_uV Smallest voltage consumers may set.

max_uV Largest voltage consumers may set.

uV_offset Offset applied to voltages from consumer to compensate for voltage drops.

min_uA Smallest current consumers may set.

max_uA Largest current consumers may set.

ilim_uA Maximum input current.

system_load Load that isn't captured by any consumer requests.

max_spread Max possible spread between coupled regulators

valid_modes_mask Mask of modes which may be configured by consumers.

valid_ops_mask Operations which may be performed by consumers.

input_uV Input voltage for regulator when supplied by another regulator.

state_disk State for regulator when system is suspended in disk mode.

state_mem State for regulator when system is suspended in mem mode.

state_standby State for regulator when system is suspended in standby mode.

initial_state Suspend state to set by default.

initial_mode Mode to set at startup.

ramp_delay Time to settle down after voltage change (unit: uV/us)

settling_time Time to settle down after voltage change when voltage change is non-linear (unit: microseconds).

settling_time_up Time to settle down after voltage increase when voltage change is non-linear (unit: microseconds).

settling_time_down Time to settle down after voltage decrease when voltage change is non-linear (unit: microseconds).

enable_time Turn-on time of the rails (unit: microseconds)

active_discharge Enable/disable active discharge. The enum regulator_active_discharge values are used for initialisation.

always_on Set if the regulator should never be disabled.

boot_on Set if the regulator is enabled when the system is initially started. If the regulator is not enabled by the hardware or bootloader then it will be enabled when the constraints are applied.

apply_uV Apply the voltage constraint when initialising.

ramp_disable Disable ramp delay when initialising or when setting voltage.

soft_start Enable soft start so that voltage ramps slowly.

pull_down Enable pull down when regulator is disabled.

over_current_protection Auto disable on over current event.

Description

This struct describes regulator and board/machine specific constraints.

struct **regulator_consumer_supply**
supply -> device mapping

Definition

```
struct regulator_consumer_supply {
  const char *dev_name;
  const char *supply;
};
```

Members

dev_name Result of dev_name() for the consumer.

supply Name for the supply.

Description

This maps a supply name to a device. Use of dev_name allows support for buses which make struct device available late such as I2C.

struct **regulator_init_data**
regulator platform initialisation data.

Definition

```
struct regulator_init_data {
  const char *supply_regulator;
  struct regulation_constraints constraints;
  int num_consumer_supplies;
  struct regulator_consumer_supply *consumer_supplies;
  int (*regulator_init)(void *driver_data);
  void *driver_data;
};
```

Members

supply_regulator Parent regulator. Specified using the regulator name as it appears in the name field in sysfs, which can be explicitly set using the constraints field 'name'.

constraints Constraints. These must be specified for the regulator to be usable.

num_consumer_supplies Number of consumer device supplies.

consumer_supplies Consumer device supply configuration.

regulator_init Callback invoked when the regulator has been registered.

driver_data Data passed to regulator_init.

Description

Initialisation constraints, our supply and consumers supplies.

struct **regulator_ops**
regulator operations.

Definition

```
struct regulator_ops {
  int (*list_voltage) (struct regulator_dev *, unsigned selector);
  int (*set_voltage) (struct regulator_dev *, int min_uV, int max_uV,␣
↪unsigned *selector);
  int (*map_voltage)(struct regulator_dev *, int min_uV, int max_uV);
  int (*set_voltage_sel) (struct regulator_dev *, unsigned selector);
  int (*get_voltage) (struct regulator_dev *);
  int (*get_voltage_sel) (struct regulator_dev *);
  int (*set_current_limit) (struct regulator_dev *, int min_uA, int max_
↪uA);
  int (*get_current_limit) (struct regulator_dev *);
  int (*set_input_current_limit) (struct regulator_dev *, int lim_uA);
  int (*set_over_current_protection) (struct regulator_dev *);
  int (*set_active_discharge) (struct regulator_dev *, bool enable);
  int (*enable) (struct regulator_dev *);
  int (*disable) (struct regulator_dev *);
  int (*is_enabled) (struct regulator_dev *);
  int (*set_mode) (struct regulator_dev *, unsigned int mode);
  unsigned int (*get_mode) (struct regulator_dev *);
  int (*get_error_flags)(struct regulator_dev *, unsigned int *flags);
  int (*enable_time) (struct regulator_dev *);
  int (*set_ramp_delay) (struct regulator_dev *, int ramp_delay);
  int (*set_voltage_time) (struct regulator_dev *, int old_uV, int new_uV);
  int (*set_voltage_time_sel) (struct regulator_dev *,unsigned int old_
↪selector, unsigned int new_selector);
  int (*set_soft_start) (struct regulator_dev *);
  int (*get_status)(struct regulator_dev *);
  unsigned int (*get_optimum_mode) (struct regulator_dev *, int input_uV,␣
↪int output_uV, int load_uA);
  int (*set_load)(struct regulator_dev *, int load_uA);
  int (*set_bypass)(struct regulator_dev *dev, bool enable);
  int (*get_bypass)(struct regulator_dev *dev, bool *enable);
  int (*set_suspend_voltage) (struct regulator_dev *, int uV);
  int (*set_suspend_enable) (struct regulator_dev *);
  int (*set_suspend_disable) (struct regulator_dev *);
  int (*set_suspend_mode) (struct regulator_dev *, unsigned int mode);
  int (*resume)(struct regulator_dev *rdev);
  int (*set_pull_down) (struct regulator_dev *);
};
```

Members

list_voltage Return one of the supported voltages, in microvolts; zero if the selector indicates a voltage that is unusable on this system; or negative errno. Selectors range from zero to one less than regulator_desc.n_voltages. Voltages may be reported in any order.

set_voltage Set the voltage for the regulator within the range specified. The driver should select the voltage closest to min_uV.

map_voltage Convert a voltage into a selector

set_voltage_sel Set the voltage for the regulator using the specified selector.

get_voltage Return the currently configured voltage for the regulator; return -ENOTRECOVERABLE if regulator can't be read at bootup and hasn't been set yet.

get_voltage_sel Return the currently configured voltage selector for the regulator; return -ENOTRECOVERABLE if regulator can't be read at bootup and hasn't been set yet.

set_current_limit Configure a limit for a current-limited regulator. The driver should select the current closest to max_uA.

get_current_limit Get the configured limit for a current-limited regulator.

set_input_current_limit Configure an input limit.

set_over_current_protection Support capability of automatically shutting down when detecting an over current event.

set_active_discharge Set active discharge enable/disable of regulators.

enable Configure the regulator as enabled.

disable Configure the regulator as disabled.

is_enabled Return 1 if the regulator is enabled, 0 if not. May also return negative errno.

set_mode Set the configured operating mode for the regulator.

get_mode Get the configured operating mode for the regulator.

get_error_flags Get the current error(s) for the regulator.

enable_time Time taken for the regulator voltage output voltage to stabilise after being enabled, in microseconds.

set_ramp_delay Set the ramp delay for the regulator. The driver should select ramp delay equal to or less than(closest) ramp_delay.

set_voltage_time Time taken for the regulator voltage output voltage to stabilise after being set to a new value, in microseconds. The function receives the from and to voltage as input, it should return the worst case.

set_voltage_time_sel Time taken for the regulator voltage output voltage to stabilise after being set to a new value, in microseconds. The function receives the from and to voltage selector as input, it should return the worst case.

set_soft_start Enable soft start for the regulator.

get_status Return actual (not as-configured) status of regulator, as a REGULATOR_STATUS value (or negative errno)

get_optimum_mode Get the most efficient operating mode for the regulator when running with the specified parameters.

set_load Set the load for the regulator.

set_bypass Set the regulator in bypass mode.

get_bypass Get the regulator bypass mode state.

set_suspend_voltage Set the voltage for the regulator when the system is suspended.

set_suspend_enable Mark the regulator as enabled when the system is suspended.

set_suspend_disable Mark the regulator as disabled when the system is suspended.

set_suspend_mode Set the operating mode for the regulator when the system is suspended.

set_pull_down Configure the regulator to pull down when the regulator is disabled.

Description

This struct describes regulator operations which can be implemented by regulator chip drivers.

struct **regulator_desc**
Static regulator descriptor

Definition

```
struct regulator_desc {
  const char *name;
  const char *supply_name;
  const char *of_match;
  const char *regulators_node;
  int (*of_parse_cb)(struct device_node *,const struct regulator_desc *,␣
↪struct regulator_config *);
  int id;
  unsigned int continuous_voltage_range:1;
  unsigned n_voltages;
  unsigned int n_current_limits;
  const struct regulator_ops *ops;
  int irq;
  enum regulator_type type;
  struct module *owner;
  unsigned int min_uV;
  unsigned int uV_step;
  unsigned int linear_min_sel;
  int fixed_uV;
  unsigned int ramp_delay;
  int min_dropout_uV;
  const struct linear_range *linear_ranges;
  const unsigned int *linear_range_selectors;
  int n_linear_ranges;
  const unsigned int *volt_table;
  const unsigned int *curr_table;
  unsigned int vsel_range_reg;
  unsigned int vsel_range_mask;
  unsigned int vsel_reg;
  unsigned int vsel_mask;
  unsigned int vsel_step;
  unsigned int csel_reg;
  unsigned int csel_mask;
  unsigned int apply_reg;
  unsigned int apply_bit;
  unsigned int enable_reg;
  unsigned int enable_mask;
  unsigned int enable_val;
  unsigned int disable_val;
```

(continues on next page)

(continued from previous page)

```
  bool enable_is_inverted;
  unsigned int bypass_reg;
  unsigned int bypass_mask;
  unsigned int bypass_val_on;
  unsigned int bypass_val_off;
  unsigned int active_discharge_on;
  unsigned int active_discharge_off;
  unsigned int active_discharge_mask;
  unsigned int active_discharge_reg;
  unsigned int soft_start_reg;
  unsigned int soft_start_mask;
  unsigned int soft_start_val_on;
  unsigned int pull_down_reg;
  unsigned int pull_down_mask;
  unsigned int pull_down_val_on;
  unsigned int enable_time;
  unsigned int off_on_delay;
  unsigned int (*of_map_mode)(unsigned int mode);
};
```

Members

name Identifying name for the regulator.

supply_name Identifying the regulator supply

of_match Name used to identify regulator in DT.

regulators_node Name of node containing regulator definitions in DT.

of_parse_cb Optional callback called only if of_match is present. Will be called for each regulator parsed from DT, during init_data parsing. The regulator_config passed as argument to the callback will be a copy of config passed to regulator_register, valid only for this particular call. Callback may freely change the config but it cannot store it for later usage. Callback should return 0 on success or negative ERRNO indicating failure.

id Numerical identifier for the regulator.

continuous_voltage_range Indicates if the regulator can set any voltage within constrains range.

n_voltages Number of selectors available for ops.list_voltage().

n_current_limits Number of selectors available for current limits

ops Regulator operations table.

irq Interrupt number for the regulator.

type Indicates if the regulator is a voltage or current regulator.

owner Module providing the regulator, used for refcounting.

min_uV Voltage given by the lowest selector (if linear mapping)

uV_step Voltage increase with each selector (if linear mapping)

linear_min_sel Minimal selector for starting linear mapping

fixed_uV Fixed voltage of rails.

ramp_delay Time to settle down after voltage change (unit: uV/us)

min_dropout_uV The minimum dropout voltage this regulator can handle

linear_ranges A constant table of possible voltage ranges.

linear_range_selectors A constant table of voltage range selectors. If pickable ranges are used each range must have corresponding selector here.

n_linear_ranges Number of entries in the **linear_ranges** (and in linear_range_selectors if used) table(s).

volt_table Voltage mapping table (if table based mapping)

curr_table Current limit mapping table (if table based mapping)

vsel_range_reg Register for range selector when using pickable ranges and `regulator_map_*_voltage_*_pickable` functions.

vsel_range_mask Mask for register bitfield used for range selector

vsel_reg Register for selector when using `regulator_map_*_voltage_*`

vsel_mask Mask for register bitfield used for selector

vsel_step Specify the resolution of selector stepping when setting voltage. If 0, then no stepping is done (requested selector is set directly), if >0 then the regulator API will ramp the voltage up/down gradually each time increasing/decreasing the selector by the specified step value.

csel_reg Register for current limit selector using regmap set_current_limit

csel_mask Mask for register bitfield used for current limit selector

apply_reg Register for initiate voltage change on the output when using regulator_set_voltage_sel_regmap

apply_bit Register bitfield used for initiate voltage change on the output when using regulator_set_voltage_sel_regmap

enable_reg Register for control when using regmap enable/disable ops

enable_mask Mask for control when using regmap enable/disable ops

enable_val Enabling value for control when using regmap enable/disable ops

disable_val Disabling value for control when using regmap enable/disable ops

enable_is_inverted A flag to indicate set enable_mask bits to disable when using regulator_enable_regmap and friends APIs.

bypass_reg Register for control when using regmap set_bypass

bypass_mask Mask for control when using regmap set_bypass

bypass_val_on Enabling value for control when using regmap set_bypass

bypass_val_off Disabling value for control when using regmap set_bypass

active_discharge_on Disabling value for control when using regmap set_active_discharge

active_discharge_off Enabling value for control when using regmap set_active_discharge

active_discharge_mask Mask for control when using regmap set_active_discharge

active_discharge_reg Register for control when using regmap set_active_discharge

soft_start_reg Register for control when using regmap set_soft_start

soft_start_mask Mask for control when using regmap set_soft_start

soft_start_val_on Enabling value for control when using regmap set_soft_start

pull_down_reg Register for control when using regmap set_pull_down

pull_down_mask Mask for control when using regmap set_pull_down

pull_down_val_on Enabling value for control when using regmap set_pull_down

enable_time Time taken for initial enable of regulator (in uS).

off_on_delay guard time (in uS), before re-enabling a regulator

of_map_mode Maps a hardware mode defined in a DeviceTree to a standard mode

Description

Each regulator registered with the core is described with a structure of this type and a struct regulator_config. This structure contains the non-varying parts of the regulator description.

struct **regulator_config**
Dynamic regulator descriptor

Definition

```
struct regulator_config {
  struct device *dev;
  const struct regulator_init_data *init_data;
  void *driver_data;
  struct device_node *of_node;
  struct regmap *regmap;
  struct gpio_desc *ena_gpiod;
};
```

Members

dev struct device for the regulator

init_data platform provided init data, passed through by driver

driver_data private regulator data

of_node OpenFirmware node to parse for device tree bindings (may be NULL).

regmap regmap to use for core regmap helpers if dev_get_regmap() is insufficient.

ena_gpiod GPIO controlling regulator enable.

Description

Each regulator registered with the core is described with a structure of this type and a struct regulator_desc. This structure contains the runtime variable parts of the regulator description.

void **regulator_lock**(struct regulator_dev * rdev)
lock a single regulator

Parameters

struct regulator_dev * rdev regulator source

Description

This function can be called many times by one task on a single regulator and its mutex will be locked only once. If a task, which is calling this function is other than the one, which initially locked the mutex, it will wait on mutex.

void **regulator_unlock**(struct regulator_dev * rdev)
unlock a single regulator

Parameters

struct regulator_dev * rdev regulator_source

Description

This function unlocks the mutex when the reference counter reaches 0.

struct regulator * **regulator_get**(struct device * dev, const char * id)
lookup and obtain a reference to a regulator.

Parameters

struct device * dev device for regulator "consumer"

const char * id Supply name or regulator ID.

Description

Returns a struct regulator corresponding to the regulator producer, or IS_ERR() condition containing errno.

Use of supply names configured via regulator_set_device_supply() is strongly encouraged. It is recommended that the supply name used should match the name used for the supply and/or the relevant device pins in the datasheet.

struct regulator * **regulator_get_exclusive**(struct device * dev, const char * id)
obtain exclusive access to a regulator.

Parameters

struct device * dev device for regulator "consumer"

const char * id Supply name or regulator ID.

Description

Returns a struct regulator corresponding to the regulator producer, or IS_ERR() condition containing errno. Other consumers will be unable to obtain this regulator while this reference is held and the use count for the regulator will be initialised to reflect the current state of the regulator.

This is intended for use by consumers which cannot tolerate shared use of the regulator such as those which need to force the regulator off for correct operation of the hardware they are controlling.

Use of supply names configured via regulator_set_device_supply() is strongly encouraged. It is recommended that the supply name used should match the name used for the supply and/or the relevant device pins in the datasheet.

struct regulator * **regulator_get_optional**(struct device * dev, const char * id)
obtain optional access to a regulator.

Parameters

struct device * dev device for regulator "consumer"

const char * id Supply name or regulator ID.

Description

Returns a struct regulator corresponding to the regulator producer, or IS_ERR() condition containing errno.

This is intended for use by consumers for devices which can have some supplies unconnected in normal use, such as some MMC devices. It can allow the regulator core to provide stub supplies for other supplies requested using normal `regulator_get()` calls without disrupting the operation of drivers that can handle absent supplies.

Use of supply names configured via regulator_set_device_supply() is strongly encouraged. It is recommended that the supply name used should match the name used for the supply and/or the relevant device pins in the datasheet.

void **regulator_put**(struct regulator * regulator)
"free" the regulator source

Parameters

struct regulator * regulator regulator source

Note

drivers must ensure that all regulator_enable calls made on this regulator source are balanced by regulator_disable calls prior to calling this function.

int **regulator_register_supply_alias**(struct device * dev, const char * id, struct device * alias_dev, const char * alias_id)
Provide device alias for supply lookup

Parameters

struct device * dev device that will be given as the regulator "consumer"

const char * id Supply name or regulator ID

struct device * alias_dev device that should be used to lookup the supply

const char * alias_id Supply name or regulator ID that should be used to lookup the supply

Description

All lookups for id on dev will instead be conducted for alias_id on alias_dev.

void **regulator_unregister_supply_alias**(struct device * dev, const char * id)
Remove device alias

Parameters

struct device * dev device that will be given as the regulator "consumer"

const char * id Supply name or regulator ID

Description

Remove a lookup alias if one exists for id on dev.

int **regulator_bulk_register_supply_alias**(struct device * dev, const char *const * id, struct device * alias_dev, const char *const * alias_id, int num_id)
register multiple aliases

Parameters

struct device * dev device that will be given as the regulator "consumer"

const char *const * id List of supply names or regulator IDs

struct device * alias_dev device that should be used to lookup the supply

const char *const * alias_id List of supply names or regulator IDs that should be used to lookup the supply

int num_id Number of aliases to register

Description

return 0 on success, an errno on failure.

This helper function allows drivers to register several supply aliases in one operation. If any of the aliases cannot be registered any aliases that were registered will be removed before returning to the caller.

void **regulator_bulk_unregister_supply_alias**(struct device * dev, const char *const * id, int num_id)
unregister multiple aliases

Parameters

struct device * dev device that will be given as the regulator "consumer"

const char *const * id List of supply names or regulator IDs

int num_id Number of aliases to unregister

Description

This helper function allows drivers to unregister several supply aliases in one operation.

int **regulator_enable**(struct regulator * regulator)
enable regulator output

Parameters

struct regulator * regulator regulator source

Description

Request that the regulator be enabled with the regulator output at the predefined voltage or current value. Calls to `regulator_enable()` must be balanced with calls to `regulator_disable()`.

NOTE

the output value can be set by other drivers, boot loader or may be hardwired in the regulator.

int **regulator_disable**(struct regulator * regulator)
disable regulator output

Parameters

struct regulator * regulator regulator source

Description

Disable the regulator output voltage or current. Calls to `regulator_enable()` must be balanced with calls to `regulator_disable()`.

NOTE

this will only disable the regulator output if no other consumer devices have it enabled, the regulator device supports disabling and machine constraints permit this operation.

int **regulator_force_disable**(struct regulator * regulator)
force disable regulator output

Parameters

struct regulator * regulator regulator source

Description

Forcibly disable the regulator output voltage or current.

NOTE

this will disable the regulator output even if other consumer devices have it enabled. This should be used for situations when device damage will likely occur if the regulator is not disabled (e.g. over temp).

int **regulator_disable_deferred**(struct regulator * regulator, int ms)
disable regulator output with delay

Parameters

struct regulator * regulator regulator source

int ms milliseconds until the regulator is disabled

Description

Execute `regulator_disable()` on the regulator after a delay. This is intended for use with devices that require some time to quiesce.

NOTE

this will only disable the regulator output if no other consumer devices have it enabled, the regulator device supports disabling and machine constraints permit this operation.

int **regulator_is_enabled**(struct regulator * regulator)
is the regulator output enabled

Parameters

struct regulator * regulator regulator source

Description

Returns positive if the regulator driver backing the source/client has requested that the device be enabled, zero if it hasn't, else a negative errno code.

Note that the device backing this regulator handle can have multiple users, so it might be enabled even if `regulator_enable()` was never called for this particular source.

int **regulator_count_voltages**(struct regulator * regulator)
count `regulator_list_voltage()` selectors

Parameters

struct regulator * regulator regulator source

Description

Returns number of selectors, or negative errno. Selectors are numbered starting at zero, and typically correspond to bitfields in hardware registers.

int **regulator_list_voltage**(struct regulator * regulator, unsigned selector)
enumerate supported voltages

Parameters

struct regulator * regulator regulator source

unsigned selector identify voltage to list

Context

can sleep

Description

Returns a voltage that can be passed to **regulator_set_voltage()**, zero if this selector code can't be used on this system, or a negative errno.

int **regulator_get_hardware_vsel_register**(struct regulator * regulator, unsigned * vsel_reg, unsigned * vsel_mask)
get the HW voltage selector register

Parameters

struct regulator * regulator regulator source

unsigned * vsel_reg voltage selector register, output parameter

unsigned * vsel_mask mask for voltage selector bitfield, output parameter

Description

Returns the hardware register offset and bitmask used for setting the regulator voltage. This might be useful when configuring voltage-scaling hardware or firmware that can make I2C requests behind the kernel's back, for example.

On success, the output parameters **vsel_reg** and **vsel_mask** are filled in and 0 is returned, otherwise a negative errno is returned.

int **regulator_list_hardware_vsel**(struct regulator * regulator, unsigned selector)
get the HW-specific register value for a selector

Parameters

struct regulator * regulator regulator source

unsigned selector identify voltage to list

Description

Converts the selector to a hardware-specific voltage selector that can be directly written to the regulator registers. The address of the voltage register can be determined by calling **regulator_get_hardware_vsel_register**.

On error a negative errno is returned.

unsigned int **regulator_get_linear_step**(struct regulator * regulator)
return the voltage step size between VSEL values

Parameters

struct regulator * regulator regulator source

Description

Returns the voltage step size between VSEL values for linear regulators, or return 0 if the regulator isn't a linear regulator.

int **regulator_is_supported_voltage**(struct regulator * regulator, int min_uV, int max_uV)
check if a voltage range can be supported

Parameters

struct regulator * regulator Regulator to check.

int min_uV Minimum required voltage in uV.

int max_uV Maximum required voltage in uV.

Description

Returns a boolean.

int **regulator_set_voltage**(struct regulator * regulator, int min_uV, int max_uV)
set regulator output voltage

Parameters

struct regulator * regulator regulator source

int min_uV Minimum required voltage in uV

int max_uV Maximum acceptable voltage in uV

Description

Sets a voltage regulator to the desired output voltage. This can be set during any regulator state. IOW, regulator can be disabled or enabled.

If the regulator is enabled then the voltage will change to the new value immediately otherwise if the regulator is disabled the regulator will output at the new voltage when enabled.

NOTE

If the regulator is shared between several devices then the lowest request voltage that meets the system constraints will be used. Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

int **regulator_set_voltage_time**(struct regulator * regulator, int old_uV, int new_uV)

get raise/fall time

Parameters

struct regulator * regulator regulator source

int old_uV starting voltage in microvolts

int new_uV target voltage in microvolts

Description

Provided with the starting and ending voltage, this function attempts to calculate the time in microseconds required to rise or fall to this new voltage.

int **regulator_set_voltage_time_sel**(struct regulator_dev * rdev, unsigned int old_selector, unsigned int new_selector)

get raise/fall time

Parameters

struct regulator_dev * rdev regulator source device

unsigned int old_selector selector for starting voltage

unsigned int new_selector selector for target voltage

Description

Provided with the starting and target voltage selectors, this function returns time in microseconds required to rise or fall to this new voltage

Drivers providing ramp_delay in regulation_constraints can use this as their set_voltage_time_sel() operation.

int **regulator_sync_voltage**(struct regulator * regulator)

re-apply last regulator output voltage

Parameters

struct regulator * regulator regulator source

Description

Re-apply the last configured voltage. This is intended to be used where some external control source the consumer is cooperating with has caused the configured voltage to change.

int **regulator_get_voltage**(struct regulator * regulator)
get regulator output voltage

Parameters

struct regulator * regulator regulator source

Description

This returns the current regulator voltage in uV.

NOTE

If the regulator is disabled it will return the voltage value. This function should not be used to determine regulator state.

int **regulator_set_current_limit**(struct regulator * regulator, int min_uA,
int max_uA)
set regulator output current limit

Parameters

struct regulator * regulator regulator source

int min_uA Minimum supported current in uA

int max_uA Maximum supported current in uA

Description

Sets current sink to the desired output current. This can be set during any regulator state. IOW, regulator can be disabled or enabled.

If the regulator is enabled then the current will change to the new value immediately otherwise if the regulator is disabled the regulator will output at the new current when enabled.

NOTE

Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

int **regulator_get_current_limit**(struct regulator * regulator)
get regulator output current

Parameters

struct regulator * regulator regulator source

Description

This returns the current supplied by the specified current sink in uA.

NOTE

If the regulator is disabled it will return the current value. This function should not be used to determine regulator state.

int **regulator_set_mode**(struct regulator * regulator, unsigned int mode)
set regulator operating mode

Parameters

struct regulator * regulator regulator source

unsigned int mode operating mode - one of the REGULATOR_MODE constants

Description

Set regulator operating mode to increase regulator efficiency or improve regulation performance.

NOTE

Regulator system constraints must be set for this regulator before calling this function otherwise this call will fail.

unsigned int **regulator_get_mode**(struct regulator * regulator)
get regulator operating mode

Parameters

struct regulator * regulator regulator source

Description

Get the current regulator operating mode.

int **regulator_get_error_flags**(struct regulator * regulator, unsigned int * flags)
get regulator error information

Parameters

struct regulator * regulator regulator source

unsigned int * flags pointer to store error flags

Description

Get the current regulator error information.

int **regulator_set_load**(struct regulator * regulator, int uA_load)
set regulator load

Parameters

struct regulator * regulator regulator source

int uA_load load current

Description

Notifies the regulator core of a new device load. This is then used by DRMS (if enabled by constraints) to set the most efficient regulator operating mode for the new regulator loading.

Consumer devices notify their supply regulator of the maximum power they will require (can be taken from device datasheet in the power consumption tables) when they change operational status and hence power state. Examples of operational state changes that can affect power consumption are :-

o Device is opened / closed. o Device I/O is about to begin or has just finished. o Device is idling in between work.

This information is also exported via sysfs to userspace.

DRMS will sum the total requested load on the regulator and change to the most efficient operating mode if platform constraints allow.

If a regulator is an always-on regulator then an individual consumer's load will still be removed if that consumer is fully disabled.

On error a negative errno is returned.

NOTE

when a regulator consumer requests to have a regulator disabled then any load that consumer requested no longer counts toward the total requested load. If the regulator is re-enabled then the previously requested load will start counting again.

int **regulator_allow_bypass**(struct regulator * regulator, bool enable)
allow the regulator to go into bypass mode

Parameters

struct regulator * regulator Regulator to configure

bool enable enable or disable bypass mode

Description

Allow the regulator to go into bypass mode if all other consumers for the regulator also enable bypass mode and the machine constraints allow this. Bypass mode means that the regulator is simply passing the input directly to the output with no regulation.

int **regulator_register_notifier**(struct regulator * regulator, struct notifier_block * nb)
register regulator event notifier

Parameters

struct regulator * regulator regulator source

struct notifier_block * nb notifier block

Description

Register notifier block to receive regulator events.

int **regulator_unregister_notifier**(struct regulator * regulator, struct notifier_block * nb)
unregister regulator event notifier

Parameters

struct regulator * regulator regulator source

struct notifier_block * nb notifier block

Description

Unregister regulator event notifier block.

int **regulator_bulk_get**(struct device * dev, int num_consumers, struct regulator_bulk_data * consumers)
get multiple regulator consumers

Parameters

struct device * dev Device to supply

int num_consumers Number of consumers to register

struct regulator_bulk_data * consumers Configuration of consumers; clients are stored here.

Description

return 0 on success, an errno on failure.

This helper function allows drivers to get several regulator consumers in one operation. If any of the regulators cannot be acquired then any regulators that were allocated will be freed before returning to the caller.

int **regulator_bulk_enable**(int num_consumers, struct regulator_bulk_data * consumers)
enable multiple regulator consumers

Parameters

int num_consumers Number of consumers

struct regulator_bulk_data * consumers Consumer data; clients are stored here. **return** 0 on success, an errno on failure

Description

This convenience API allows consumers to enable multiple regulator clients in a single API call. If any consumers cannot be enabled then any others that were enabled will be disabled again prior to return.

int **regulator_bulk_disable**(int num_consumers, struct regulator_bulk_data * consumers)
disable multiple regulator consumers

Parameters

int num_consumers Number of consumers

struct regulator_bulk_data * consumers Consumer data; clients are stored here. **return** 0 on success, an errno on failure

Description

This convenience API allows consumers to disable multiple regulator clients in a single API call. If any consumers cannot be disabled then any others that were disabled will be enabled again prior to return.

int **regulator_bulk_force_disable**(int num_consumers, struct regulator_bulk_data * consumers)
force disable multiple regulator consumers

Parameters

int num_consumers Number of consumers

struct regulator_bulk_data * consumers Consumer data; clients are stored here. **return** 0 on success, an errno on failure

Description

This convenience API allows consumers to forcibly disable multiple regulator clients in a single API call.

NOTE

This should be used for situations when device damage will likely occur if the regulators are not disabled (e.g. over temp). Although regulator_force_disable function call for some consumers can return error numbers, the function is called for all consumers.

void **regulator_bulk_free**(int num_consumers, struct regulator_bulk_data * consumers)
free multiple regulator consumers

Parameters

int num_consumers Number of consumers

struct regulator_bulk_data * consumers Consumer data; clients are stored here.

Description

This convenience API allows consumers to free multiple regulator clients in a single API call.

int **regulator_notifier_call_chain**(struct regulator_dev * rdev, unsigned long event, void * data)
call regulator event notifier

Parameters

struct regulator_dev * rdev regulator source

unsigned long event notifier block

void * data callback-specific data.

Description

Called by regulator drivers to notify clients a regulator event has occurred. We also notify regulator clients downstream. Note lock must be held by caller.

int **regulator_mode_to_status**(unsigned int mode)
convert a regulator mode into a status

Parameters

unsigned int mode Mode to convert

Description

Convert a regulator mode into a status.

struct regulator_dev * **regulator_register**(const struct regulator_desc * regulator_desc, const struct regulator_config * cfg)
register regulator

Parameters

const struct regulator_desc * regulator_desc regulator to register

const struct regulator_config * cfg runtime configuration for regulator

Description

Called by regulator drivers to register a regulator. Returns a valid pointer to struct regulator_dev on success or an ERR_PTR() on error.

void **regulator_unregister**(struct regulator_dev * rdev)
unregister regulator

Parameters

struct regulator_dev * rdev regulator to unregister

Description

Called by regulator drivers to unregister a regulator.

void **regulator_has_full_constraints**(void)
the system has fully specified constraints

Parameters

void no arguments

Description

Calling this function will cause the regulator API to disable all regulators which have a zero use count and don’t have an always_on constraint in a late_initcall.

The intention is that this will become the default behaviour in a future kernel release so users are encouraged to use this facility now.

void * **rdev_get_drvdata**(struct regulator_dev * rdev)
get rdev regulator driver data

Parameters

struct regulator_dev * rdev regulator

Description

Get rdev regulator driver private data. This call can be used in the regulator driver context.

void * **regulator_get_drvdata**(struct regulator * regulator)
get regulator driver data

Parameters

struct regulator * regulator regulator

Description

Get regulator driver private data. This call can be used in the consumer driver context when non API regulator specific functions need to be called.

void **regulator_set_drvdata**(struct regulator * regulator, void * data)
set regulator driver data

Parameters

struct regulator * regulator regulator

void * data data

int **rdev_get_id**(struct regulator_dev * rdev)
get regulator ID

Parameters

struct regulator_dev * rdev regulator

CHAPTER

EIGHTEEN

INDUSTRIAL I/O

Contents:

18.1 Introduction

The main purpose of the Industrial I/O subsystem (IIO) is to provide support for devices that in some sense perform either analog-to-digital conversion (ADC) or digital-to-analog conversion (DAC) or both. The aim is to fill the gap between the somewhat similar hwmon and input subsystems. Hwmon is directed at low sample rate sensors used to monitor and control the system itself, like fan speed control or temperature measurement. Input is, as its name suggests, focused on human interaction input devices (keyboard, mouse, touchscreen). In some cases there is considerable overlap between these and IIO.

Devices that fall into this category include:

- analog to digital converters (ADCs)
- accelerometers
- capacitance to digital converters (CDCs)
- digital to analog converters (DACs)
- gyroscopes
- inertial measurement units (IMUs)
- color and light sensors
- magnetometers
- pressure sensors
- proximity sensors
- temperature sensors

Usually these sensors are connected via SPI or I2C. A common use case of the sensors devices is to have combined functionality (e.g. light plus proximity sensor).

18.2 Core elements

The Industrial I/O core offers both a unified framework for writing drivers for many different types of embedded sensors and a standard interface to user space applications manipulating sensors. The implementation can be found under `drivers/iio/industrialio-*`

18.2.1 Industrial I/O Devices

- struct `iio_dev` - industrial I/O device
- `iio_device_alloc()` - allocate an `iio_dev` from a driver
- `iio_device_free()` - free an `iio_dev` from a driver
- `iio_device_register()` - register a device with the IIO subsystem
- `iio_device_unregister()` - unregister a device from the IIO subsystem

An IIO device usually corresponds to a single hardware sensor and it provides all the information needed by a driver handling a device. Let's first have a look at the functionality embedded in an IIO device then we will show how a device driver makes use of an IIO device.

There are two ways for a user space application to interact with an IIO driver.

1. `/sys/bus/iio/iio:deviceX/`, this represents a hardware sensor and groups together the data channels of the same chip.
2. `/dev/iio:deviceX`, character device node interface used for buffered data transfer and for events information retrieval.

A typical IIO driver will register itself as an I2C or SPI driver and will create two routines, probe and remove.

At probe:

1. Call `iio_device_alloc()`, which allocates memory for an IIO device.
2. Initialize IIO device fields with driver specific information (e.g. device name, device channels).
3. Call `iio_device_register()`, this registers the device with the IIO core. After this call the device is ready to accept requests from user space applications.

At remove, we free the resources allocated in probe in reverse order:

1. `iio_device_unregister()`, unregister the device from the IIO core.
2. `iio_device_free()`, free the memory allocated for the IIO device.

IIO device sysfs interface

Attributes are sysfs files used to expose chip info and also allowing applications to set various configuration parameters. For device with index X, attributes can be found under /sys/bus/iio/iio:deviceX/ directory. Common attributes are:

- `name`, description of the physical chip.
- `dev`, shows the major:minor pair associated with `/dev/iio:deviceX` node.
- `sampling_frequency_available`, available discrete set of sampling frequency values for device.
- Available standard attributes for IIO devices are described in the `Documentation/ABI/testing/sysfs-bus-iio` file in the Linux kernel sources.

IIO device channels

struct `iio_chan_spec` - specification of a single channel

An IIO device channel is a representation of a data channel. An IIO device can have one or multiple channels. For example:

- a thermometer sensor has one channel representing the temperature measurement.
- a light sensor with two channels indicating the measurements in the visible and infrared spectrum.
- an accelerometer can have up to 3 channels representing acceleration on X, Y and Z axes.

An IIO channel is described by the struct `iio_chan_spec`. A thermometer driver for the temperature sensor in the example above would have to describe its channel as follows:

```
static const struct iio_chan_spec temp_channel[] = {
     {
          .type = IIO_TEMP,
          .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED),
     },
};
```

Channel sysfs attributes exposed to userspace are specified in the form of bitmasks. Depending on their shared info, attributes can be set in one of the following masks:

- **info_mask_separate**, attributes will be specific to this channel
- **info_mask_shared_by_type**, attributes are shared by all channels of the same type
- **info_mask_shared_by_dir**, attributes are shared by all channels of the same direction
- **info_mask_shared_by_all**, attributes are shared by all channels

When there are multiple data channels per channel type we have two ways to distinguish between them:

- set **.modified** field of `iio_chan_spec` to 1. Modifiers are specified using **.channel2** field of the same `iio_chan_spec` structure and are used to indicate a physically unique characteristic of the channel such as its direction or spectral response. For example, a light sensor can have two channels, one for infrared light and one for both infrared and visible light.
- set **.indexed** field of `iio_chan_spec` to 1. In this case the channel is simply another instance with an index specified by the **.channel** field.

Here is how we can make use of the channel's modifiers:

```
static const struct iio_chan_spec light_channels[] = {
        {
                .type = IIO_INTENSITY,
                .modified = 1,
                .channel2 = IIO_MOD_LIGHT_IR,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
                .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
        },
        {
                .type = IIO_INTENSITY,
                .modified = 1,
                .channel2 = IIO_MOD_LIGHT_BOTH,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
                .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
        },
        {
                .type = IIO_LIGHT,
                .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED),
                .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
        },
   }
```

This channel's definition will generate two separate sysfs files for raw data retrieval:

- `/sys/bus/iio/iio:deviceX/in_intensity_ir_raw`
- `/sys/bus/iio/iio:deviceX/in_intensity_both_raw`

one file for processed data:

- `/sys/bus/iio/iio:deviceX/in_illuminance_input`

and one shared sysfs file for sampling frequency:

- `/sys/bus/iio/iio:deviceX/sampling_frequency`.

Here is how we can make use of the channel's indexing:

```
static const struct iio_chan_spec light_channels[] = {
        {
                .type = IIO_VOLTAGE,
                .indexed = 1,
                .channel = 0,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
```

(continues on next page)

(continued from previous page)

```
        },
        {
                .type = IIO_VOLTAGE,
                .indexed = 1,
                .channel = 1,
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        },
}
```

This will generate two separate attributes files for raw data retrieval:

- `/sys/bus/iio/devices/iio:deviceX/in_voltage0_raw`, representing voltage measurement for channel 0.
- `/sys/bus/iio/devices/iio:deviceX/in_voltage1_raw`, representing voltage measurement for channel 1.

More details

struct **iio_chan_spec_ext_info**
Extended channel info attribute

Definition

```
struct iio_chan_spec_ext_info {
  const char *name;
  enum iio_shared_by shared;
  ssize_t (*read)(struct iio_dev *, uintptr_t private, struct iio_chan_
↪spec const *, char *buf);
  ssize_t (*write)(struct iio_dev *, uintptr_t private,struct iio_chan_
↪spec const *, const char *buf, size_t len);
  uintptr_t private;
};
```

Members

name Info attribute name

shared Whether this attribute is shared between all channels.

read Read callback for this info attribute, may be NULL.

write Write callback for this info attribute, may be NULL.

private Data private to the driver.

struct **iio_enum**
Enum channel info attribute

Definition

```
struct iio_enum {
  const char * const *items;
  unsigned int num_items;
  int (*set)(struct iio_dev *, const struct iio_chan_spec *, unsigned int);
  int (*get)(struct iio_dev *, const struct iio_chan_spec *);
};
```

Members

items An array of strings.

num_items Length of the item array.

set Set callback function, may be NULL.

get Get callback function, may be NULL.

Description

The iio_enum struct can be used to implement enum style channel attributes. Enum style attributes are those which have a set of strings which map to unsigned integer values. The IIO enum helper code takes care of mapping between value and string as well as generating a "_available" file which contains a list of all available items. The set callback will be called when the attribute is updated. The last parameter is the index to the newly activated item. The get callback will be used to query the currently active item and is supposed to return the index for it.

IIO_ENUM(_name, _shared, _e)
Initialize enum extended channel attribute

Parameters

_name Attribute name

_shared Whether the attribute is shared between all channels

_e Pointer to an iio_enum struct

Description

This should usually be used together with `IIO_ENUM_AVAILABLE()`

IIO_ENUM_AVAILABLE(_name, _e)
Initialize enum available extended channel attribute

Parameters

_name Attribute name ("_available" will be appended to the name)

_e Pointer to an iio_enum struct

Description

Creates a read only attribute which lists all the available enum items in a space separated list. This should usually be used together with `IIO_ENUM()`

struct **iio_mount_matrix**
iio mounting matrix

Definition

```
struct iio_mount_matrix {
  const char *rotation[9];
};
```

Members

rotation 3 dimensional space rotation matrix defining sensor alignment with main hardware

IIO_MOUNT_MATRIX(_shared, _get)
Initialize mount matrix extended channel attribute

Parameters

_shared Whether the attribute is shared between all channels

_get Pointer to an iio_get_mount_matrix_t accessor

struct **iio_event_spec**
specification for a channel event

Definition

```
struct iio_event_spec {
  enum iio_event_type type;
  enum iio_event_direction dir;
  unsigned long mask_separate;
  unsigned long mask_shared_by_type;
  unsigned long mask_shared_by_dir;
  unsigned long mask_shared_by_all;
};
```

Members

type Type of the event

dir Direction of the event

mask_separate Bit mask of enum iio_event_info values. Attributes set in this mask will be registered per channel.

mask_shared_by_type Bit mask of enum iio_event_info values. Attributes set in this mask will be shared by channel type.

mask_shared_by_dir Bit mask of enum iio_event_info values. Attributes set in this mask will be shared by channel type and direction.

mask_shared_by_all Bit mask of enum iio_event_info values. Attributes set in this mask will be shared by all channels.

struct **iio_chan_spec**
specification of a single channel

Definition

```
struct iio_chan_spec {
  enum iio_chan_type      type;
  int channel;
  int channel2;
  unsigned long           address;
  int scan_index;
  struct {
    char sign;
    u8 realbits;
    u8 storagebits;
    u8 shift;
    u8 repeat;
    enum iio_endian endianness;
  } scan_type;
```

(continues on next page)

(continued from previous page)

```
  long info_mask_separate;
  long info_mask_separate_available;
  long info_mask_shared_by_type;
  long info_mask_shared_by_type_available;
  long info_mask_shared_by_dir;
  long info_mask_shared_by_dir_available;
  long info_mask_shared_by_all;
  long info_mask_shared_by_all_available;
  const struct iio_event_spec *event_spec;
  unsigned int            num_event_specs;
  const struct iio_chan_spec_ext_info *ext_info;
  const char              *extend_name;
  const char              *datasheet_name;
  unsigned modified:1;
  unsigned indexed:1;
  unsigned output:1;
  unsigned differential:1;
};
```

Members

type What type of measurement is the channel making.

channel What number do we wish to assign the channel.

channel2 If there is a second number for a differential channel then this is it. If modified is set then the value here specifies the modifier.

address Driver specific identifier.

scan_index Monotonic index to give ordering in scans when read from a buffer.

scan_type struct describing the scan type

scan_type.sign ‘s’ or ‘u’ to specify signed or unsigned

scan_type.realbits Number of valid bits of data

scan_type.storagebits Realbits + padding

scan_type.shift Shift right by this before masking out realbits.

scan_type.repeat Number of times real/storage bits repeats. When the repeat element is more than 1, then the type element in sysfs will show a repeat value. Otherwise, the number of repetitions is omitted.

scan_type.endianness little or big endian

info_mask_separate What information is to be exported that is specific to this channel.

info_mask_separate_available What availability information is to be exported that is specific to this channel.

info_mask_shared_by_type What information is to be exported that is shared by all channels of the same type.

info_mask_shared_by_type_available What availability information is to be exported that is shared by all channels of the same type.

info_mask_shared_by_dir What information is to be exported that is shared by all channels of the same direction.

info_mask_shared_by_dir_available What availability information is to be exported that is shared by all channels of the same direction.

info_mask_shared_by_all What information is to be exported that is shared by all channels.

info_mask_shared_by_all_available What availability information is to be exported that is shared by all channels.

event_spec Array of events which should be registered for this channel.

num_event_specs Size of the event_spec array.

ext_info Array of extended info attributes for this channel. The array is NULL terminated, the last element should have its name field set to NULL.

extend_name Allows labeling of channel attributes with an informative name. Note this has no effect codes etc, unlike modifiers.

datasheet_name A name used in in-kernel mapping of channels. It should correspond to the first name that the channel is referred to by in the datasheet (e.g. IND), or the nearest possible compound name (e.g. IND-INC).

modified Does a modifier apply to this channel. What these are depends on the channel type. Modifier is set in channel2. Examples are IIO_MOD_X for axial sensors about the 'x' axis.

indexed Specify the channel has a numerical index. If not, the channel index number will be suppressed for sysfs attributes but not for event codes.

output Channel is output.

differential Channel is differential.

bool **iio_channel_has_info**(const struct iio_chan_spec * chan, enum iio_chan_info_enum type)
Checks whether a channel supports a info attribute

Parameters

const struct iio_chan_spec * chan The channel to be queried

enum iio_chan_info_enum type Type of the info attribute to be checked

Description

Returns true if the channels supports reporting values for the given info attribute type, false otherwise.

bool **iio_channel_has_available**(const struct iio_chan_spec * chan, enum iio_chan_info_enum type)
Checks if a channel has an available attribute

Parameters

const struct iio_chan_spec * chan The channel to be queried

enum iio_chan_info_enum type Type of the available attribute to be checked

Description

Returns true if the channel supports reporting available values for the given attribute type, false otherwise.

struct **iio_info**
constant information about device

Definition

```
struct iio_info {
  const struct attribute_group    *event_attrs;
  const struct attribute_group    *attrs;
  int (*read_raw)(struct iio_dev *indio_dev,struct iio_chan_spec const␣
↪*chan,int *val,int *val2, long mask);
  int (*read_raw_multi)(struct iio_dev *indio_dev,struct iio_chan_spec␣
↪const *chan,int max_len,int *vals,int *val_len, long mask);
  int (*read_avail)(struct iio_dev *indio_dev,struct iio_chan_spec const␣
↪*chan,const int **vals,int *type,int *length, long mask);
  int (*write_raw)(struct iio_dev *indio_dev,struct iio_chan_spec const␣
↪*chan,int val,int val2, long mask);
  int (*write_raw_get_fmt)(struct iio_dev *indio_dev,struct iio_chan_spec␣
↪const *chan, long mask);
  int (*read_event_config)(struct iio_dev *indio_dev,const struct iio_chan_
↪spec *chan,enum iio_event_type type, enum iio_event_direction dir);
  int (*write_event_config)(struct iio_dev *indio_dev,const struct iio_
↪chan_spec *chan,enum iio_event_type type,enum iio_event_direction dir,␣
↪int state);
  int (*read_event_value)(struct iio_dev *indio_dev,const struct iio_chan_
↪spec *chan,enum iio_event_type type,enum iio_event_direction dir, enum␣
↪iio_event_info info, int *val, int *val2);
  int (*write_event_value)(struct iio_dev *indio_dev,const struct iio_chan_
↪spec *chan,enum iio_event_type type,enum iio_event_direction dir, enum␣
↪iio_event_info info, int val, int val2);
  int (*validate_trigger)(struct iio_dev *indio_dev, struct iio_trigger␣
↪*trig);
  int (*update_scan_mode)(struct iio_dev *indio_dev, const unsigned long␣
↪*scan_mask);
  int (*debugfs_reg_access)(struct iio_dev *indio_dev,unsigned reg,␣
↪unsigned writeval, unsigned *readval);
  int (*of_xlate)(struct iio_dev *indio_dev, const struct of_phandle_args␣
↪*iiospec);
  int (*hwfifo_set_watermark)(struct iio_dev *indio_dev, unsigned val);
  int (*hwfifo_flush_to_buffer)(struct iio_dev *indio_dev, unsigned count);
};
```

Members

event_attrs event control attributes

attrs general purpose device attributes

read_raw function to request a value from the device. mask specifies which value. Note 0 means a reading of the channel in question. Return value will specify the type of value returned by the device. val and val2 will contain the elements making up the returned value.

read_raw_multi function to return values from the device. mask specifies which value. Note 0 means a reading of the channel in question. Return value will

specify the type of value returned by the device. vals pointer contain the elements making up the returned value. max_len specifies maximum number of elements vals pointer can contain. val_len is used to return length of valid elements in vals.

read_avail function to return the available values from the device. mask specifies which value. Note 0 means the available values for the channel in question. Return value specifies if a IIO_AVAIL_LIST or a IIO_AVAIL_RANGE is returned in vals. The type of the vals are returned in type and the number of vals is returned in length. For ranges, there are always three vals returned; min, step and max. For lists, all possible values are enumerated.

write_raw function to write a value to the device. Parameters are the same as for read_raw.

write_raw_get_fmt callback function to query the expected format/precision. If not set by the driver, write_raw returns IIO_VAL_INT_PLUS_MICRO.

read_event_config find out if the event is enabled.

write_event_config set if the event is enabled.

read_event_value read a configuration value associated with the event.

write_event_value write a configuration value for the event.

validate_trigger function to validate the trigger when the current trigger gets changed.

update_scan_mode function to configure device and scan buffer when channels have changed

debugfs_reg_access function to read or write register value of device

of_xlate function pointer to obtain channel specifier index. When #iio-cells is greater than '0', the driver could provide a custom of_xlate function that reads the args and returns the appropriate index in registered IIO channels array.

hwfifo_set_watermark function pointer to set the current hardware fifo watermark level; see hwfifo_* entries in Documentation/ABI/testing/sysfs-bus-iio for details on how the hardware fifo operates

hwfifo_flush_to_buffer function pointer to flush the samples stored in the hardware fifo to the device buffer. The driver should not flush more than count samples. The function must return the number of samples flushed, 0 if no samples were flushed or a negative integer if no samples were flushed and there was an error.

struct **iio_buffer_setup_ops**
buffer setup related callbacks

Definition

```
struct iio_buffer_setup_ops {
  int (*preenable)(struct iio_dev *);
  int (*postenable)(struct iio_dev *);
  int (*predisable)(struct iio_dev *);
  int (*postdisable)(struct iio_dev *);
```

(continues on next page)

(continued from previous page)

```
  bool (*validate_scan_mask)(struct iio_dev *indio_dev, const unsigned␣
↪long *scan_mask);
};
```

Members

preenable [DRIVER] function to run prior to marking buffer enabled

postenable [DRIVER] function to run after marking buffer enabled

predisable [DRIVER] function to run prior to marking buffer disabled

postdisable [DRIVER] function to run after marking buffer disabled

validate_scan_mask [DRIVER] function callback to check whether a given scan mask is valid for the device.

struct **iio_dev**
industrial I/O device

Definition

```
struct iio_dev {
  int id;
  struct module                   *driver_module;
  int modes;
  int currentmode;
  struct device                   dev;
  struct iio_event_interface      *event_interface;
  struct iio_buffer               *buffer;
  struct list_head                buffer_list;
  int scan_bytes;
  struct mutex                    mlock;
  const unsigned long             *available_scan_masks;
  unsigned masklength;
  const unsigned long             *active_scan_mask;
  bool scan_timestamp;
  unsigned scan_index_timestamp;
  struct iio_trigger              *trig;
  bool trig_readonly;
  struct iio_poll_func            *pollfunc;
  struct iio_poll_func            *pollfunc_event;
  struct iio_chan_spec const      *channels;
  int num_channels;
  struct list_head                channel_attr_list;
  struct attribute_group          chan_attr_group;
  const char                      *name;
  const char                      *label;
  const struct iio_info           *info;
  clockid_t clock_id;
  struct mutex                    info_exist_lock;
  const struct iio_buffer_setup_ops       *setup_ops;
  struct cdev                     chrdev;
#define IIO_MAX_GROUPS 6;
  const struct attribute_group    *groups[IIO_MAX_GROUPS + 1];
  int groupcounter;
  unsigned long                   flags;
```

(continues on next page)

(continued from previous page)

```
#if defined(CONFIG_DEBUG_FS);
  struct dentry                   *debugfs_dentry;
  unsigned cached_reg_addr;
  char read_buf[20];
  unsigned int                    read_buf_len;
#endif;
};
```

Members

id [INTERN] used to identify device internally

driver_module [INTERN] used to make it harder to undercut users

modes [DRIVER] operating modes supported by device

currentmode [DRIVER] current operating mode

dev [DRIVER] device structure, should be assigned a parent and owner

event_interface [INTERN] event chrdevs associated with interrupt lines

buffer [DRIVER] any buffer present

buffer_list [INTERN] list of all buffers currently attached

scan_bytes [INTERN] num bytes captured to be fed to buffer demux

mlock [INTERN] lock used to prevent simultaneous device state changes

available_scan_masks [DRIVER] optional array of allowed bitmasks

masklength [INTERN] the length of the mask established from channels

active_scan_mask [INTERN] union of all scan masks requested by buffers

scan_timestamp [INTERN] set if any buffers have requested timestamp

scan_index_timestamp [INTERN] cache of the index to the timestamp

trig [INTERN] current device trigger (buffer modes)

trig_readonly [INTERN] mark the current trigger immutable

pollfunc [DRIVER] function run on trigger being received

pollfunc_event [DRIVER] function run on events trigger being received

channels [DRIVER] channel specification structure table

num_channels [DRIVER] number of channels specified in **channels**.

channel_attr_list [INTERN] keep track of automatically created channel attributes

chan_attr_group [INTERN] group for all attrs in base directory

name [DRIVER] name of the device.

label [DRIVER] unique name to identify which device this is

info [DRIVER] callbacks and constant info from driver

clock_id [INTERN] timestamping clock posix identifier

info_exist_lock [INTERN] lock to prevent use during removal

setup_ops [DRIVER] callbacks to call before and after buffer enable/disable

chrdev [INTERN] associated character device

groups [INTERN] attribute groups

groupcounter [INTERN] index of next attribute group

flags [INTERN] file ops related flags including busy flag.

debugfs_dentry [INTERN] device specific debugfs dentry.

cached_reg_addr [INTERN] cached register address for debugfs reads.

iio_device_register(indio_dev)
register a device with the IIO subsystem

Parameters

indio_dev Device structure filled by the device driver

devm_iio_device_register(dev, indio_dev)
Resource-managed `iio_device_register()`

Parameters

dev Device to allocate iio_dev for

indio_dev Device structure filled by the device driver

Description

Managed iio_device_register. The IIO device registered with this function is automatically unregistered on driver detach. This function calls `iio_device_register()` internally. Refer to that function for more information.

Return

0 on success, negative error number on failure.

void **iio_device_put**(struct iio_dev * indio_dev)
reference counted deallocation of struct device

Parameters

struct iio_dev * indio_dev IIO device structure containing the device

clockid_t **iio_device_get_clock**(const struct iio_dev * indio_dev)
Retrieve current timestamping clock for the device

Parameters

const struct iio_dev * indio_dev IIO device structure containing the device

struct iio_dev * **dev_to_iio_dev**(struct device * dev)
Get IIO device struct from a device struct

Parameters

struct device * dev The device embedded in the IIO device

Note

The device must be a IIO device, otherwise the result is undefined.

struct iio_dev * **iio_device_get**(struct iio_dev * indio_dev)
increment reference count for the device

Parameters

struct iio_dev * indio_dev IIO device structure

Return

The passed IIO device

void **iio_device_set_drvdata**(struct iio_dev * indio_dev, void * data)
Set device driver data

Parameters

struct iio_dev * indio_dev IIO device structure

void * data Driver specific data

Description

Allows to attach an arbitrary pointer to an IIO device, which can later be retrieved by `iio_device_get_drvdata()`.

void * **iio_device_get_drvdata**(struct iio_dev * indio_dev)
Get device driver data

Parameters

struct iio_dev * indio_dev IIO device structure

Description

Returns the data previously set with `iio_device_set_drvdata()`

bool **iio_buffer_enabled**(struct iio_dev * indio_dev)
helper function to test if the buffer is enabled

Parameters

struct iio_dev * indio_dev IIO device structure for device

struct dentry * **iio_get_debugfs_dentry**(struct iio_dev * indio_dev)
helper function to get the debugfs_dentry

Parameters

struct iio_dev * indio_dev IIO device structure for device

IIO_DEGREE_TO_RAD(deg)
Convert degree to rad

Parameters

deg A value in degree

Description

Returns the given value converted from degree to rad

IIO_RAD_TO_DEGREE(rad)
Convert rad to degree

Parameters

rad A value in rad

Description

Returns the given value converted from rad to degree

IIO_G_TO_M_S_2(g)
Convert g to meter / second**2

Parameters

g A value in g

Description

Returns the given value converted from g to meter / second**2

IIO_M_S_2_TO_G(ms2)
Convert meter / second**2 to g

Parameters

ms2 A value in meter / second**2

Description

Returns the given value converted from meter / second**2 to g

int **iio_device_set_clock**(struct iio_dev * indio_dev, clockid_t clock_id)
Set current timestamping clock for the device

Parameters

struct iio_dev * indio_dev IIO device structure containing the device

clockid_t clock_id timestamping clock posix identifier to set.

s64 **iio_get_time_ns**(const struct iio_dev * indio_dev)
utility function to get a time stamp for events etc

Parameters

const struct iio_dev * indio_dev device

unsigned int **iio_get_time_res**(const struct iio_dev * indio_dev)
utility function to get time stamp clock resolution in nano seconds.

Parameters

const struct iio_dev * indio_dev device

int **iio_read_mount_matrix**(struct device * dev, const char * propname,
struct iio_mount_matrix * matrix)
retrieve iio device mounting matrix from device “mount-matrix” property

Parameters

struct device * dev device the mounting matrix property is assigned to

const char * propname device specific mounting matrix property name

struct iio_mount_matrix * matrix where to store retrieved matrix

Description

If device is assigned no mounting matrix property, a default 3x3 identity matrix will be filled in.

Return

0 if success, or a negative error code on failure.

ssize_t **iio_format_value**(char * buf, unsigned int type, int size, int * vals)
Formats a IIO value into its string representation

Parameters

char * buf The buffer to which the formatted value gets written which is assumed to be big enough (i.e. PAGE_SIZE).

unsigned int type One of the IIO_VAL_* constants. This decides how the val and val2 parameters are formatted.

int size Number of IIO value entries contained in vals

int * vals Pointer to the values, exact meaning depends on the type parameter.

Return

0 by default, a negative number on failure or the total number of characters written for a type that belongs to the IIO_VAL_* constant.

int **iio_str_to_fixpoint**(const char * str, int fract_mult, int * integer, int * fract)
Parse a fixed-point number from a string

Parameters

const char * str The string to parse

int fract_mult Multiplier for the first decimal place, should be a power of 10

int * integer The integer part of the number

int * fract The fractional part of the number

Description

Returns 0 on success, or a negative error code if the string could not be parsed.

struct iio_dev * **iio_device_alloc**(int sizeof_priv)
allocate an iio_dev from a driver

Parameters

int sizeof_priv Space to allocate for private structure.

void **iio_device_free**(struct iio_dev * dev)
free an iio_dev from a driver

Parameters

struct iio_dev * dev the iio_dev associated with the device

struct iio_dev * **devm_iio_device_alloc**(struct device * dev, int sizeof_priv)
Resource-managed `iio_device_alloc()`

Parameters

struct device * dev Device to allocate iio_dev for

int sizeof_priv Space to allocate for private structure.

Description

Managed iio_device_alloc. iio_dev allocated with this function is automatically freed on driver detach.

Return

Pointer to allocated iio_dev on success, NULL on failure.

void **iio_device_unregister**(struct iio_dev * indio_dev)
unregister a device from the IIO subsystem

Parameters

struct iio_dev * indio_dev Device structure representing the device.

int **iio_device_claim_direct_mode**(struct iio_dev * indio_dev)
Keep device in direct mode

Parameters

struct iio_dev * indio_dev the iio_dev associated with the device

Description

If the device is in direct mode it is guaranteed to stay that way until `iio_device_release_direct_mode()` is called.

Use with `iio_device_release_direct_mode()`

Return

0 on success, -EBUSY on failure

void **iio_device_release_direct_mode**(struct iio_dev * indio_dev)
releases claim on direct mode

Parameters

struct iio_dev * indio_dev the iio_dev associated with the device

Description

Release the claim. Device is no longer guaranteed to stay in direct mode.

Use with `iio_device_claim_direct_mode()`

18.3 Buffers

- struct `iio_buffer` —general buffer structure
- `iio_validate_scan_mask_onehot()` —Validates that exactly one channel is selected
- `iio_buffer_get()` —Grab a reference to the buffer
- `iio_buffer_put()` —Release the reference to the buffer

The Industrial I/O core offers a way for continuous data capture based on a trigger source. Multiple data channels can be read at once from `/dev/iio:deviceX` character device node, thus reducing the CPU load.

18.3.1 IIO buffer sysfs interface

An IIO buffer has an associated attributes directory under `/sys/bus/iio/iio:deviceX/buffer/*`. Here are some of the existing attributes:

- `length`, the total number of data samples (capacity) that can be stored by the buffer.
- `enable`, activate buffer capture.

18.3.2 IIO buffer setup

The meta information associated with a channel reading placed in a buffer is called a scan element. The important bits configuring scan elements are exposed to userspace applications via the `/sys/bus/iio/iio:deviceX/scan_elements/*` directory. This file contains attributes of the following form:

- `enable`, used for enabling a channel. If and only if its attribute is non zero, then a triggered capture will contain data samples for this channel.
- `type`, description of the scan element data storage within the buffer and hence the form in which it is read from user space. Format is [be|le]:[s|u]bits/storagebitsXrepeat[>>shift] . * be or le, specifies big or little endian. * s or u, specifies if signed (2's complement) or unsigned. * bits, is the number of valid data bits. * storagebits, is the number of bits (after padding) that it occupies in the buffer. * shift, if specified, is the shift that needs to be applied prior to masking out unused bits. * repeat, specifies the number of bits/storagebits repetitions. When the repeat element is 0 or 1, then the repeat value is omitted.

For example, a driver for a 3-axis accelerometer with 12 bit resolution where data is stored in two 8-bits registers as follows:

```
  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
|D3 |D2 |D1 |D0 | X | X | X | X | (LOW byte, address 0x06)
+---+---+---+---+---+---+---+---+

  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
|D11|D10|D9 |D8 |D7 |D6 |D5 |D4 | (HIGH byte, address 0x07)
+---+---+---+---+---+---+---+---+
```

will have the following scan element type for each axis:

```
$ cat /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_y_type
le:s12/16>>4
```

A user space application will interpret data samples read from the buffer as two byte little endian signed data, that needs a 4 bits right shift before masking out the 12 valid bits of data.

For implementing buffer support a driver should initialize the following fields in iio_chan_spec definition:

```
struct iio_chan_spec {
/* other members */
        int scan_index
        struct {
                char sign;
                u8 realbits;
                u8 storagebits;
                u8 shift;
                u8 repeat;
                enum iio_endian endianness;
               } scan_type;
       };
```

The driver implementing the accelerometer described above will have the following channel definition:

```
struct struct iio_chan_spec accel_channels[] = {
        {
                .type = IIO_ACCEL,
                .modified = 1,
                .channel2 = IIO_MOD_X,
                /* other stuff here */
                .scan_index = 0,
                .scan_type = {
                        .sign = 's',
                        .realbits = 12,
                        .storagebits = 16,
                        .shift = 4,
                        .endianness = IIO_LE,
                },
        }
        /* similar for Y (with channel2 = IIO_MOD_Y, scan_index = 1)
         * and Z (with channel2 = IIO_MOD_Z, scan_index = 2) axis
         */
 }
```

Here **scan_index** defines the order in which the enabled channels are placed inside the buffer. Channels with a lower **scan_index** will be placed before channels with a higher index. Each channel needs to have a unique **scan_index**.

Setting **scan_index** to -1 can be used to indicate that the specific channel does not support buffered capture. In this case no entries will be created for the channel in the scan_elements directory.

18.3.3 More details

int **iio_push_to_buffers_with_timestamp**(struct iio_dev * indio_dev, void * data, int64_t timestamp)
push data and timestamp to buffers

Parameters

struct iio_dev * indio_dev iio_dev structure for device.

void * data sample data

int64_t timestamp timestamp for the sample data

Description

Pushes data to the IIO device's buffers. If timestamps are enabled for the device the function will store the supplied timestamp as the last element in the sample data buffer before pushing it to the device buffers. The sample data buffer needs to be large enough to hold the additional timestamp (usually the buffer should be indio->scan_bytes bytes large).

Returns 0 on success, a negative error code otherwise.

void **iio_buffer_set_attrs**(struct iio_buffer * buffer, const struct attribute ** attrs)
Set buffer specific attributes

Parameters

struct iio_buffer * buffer The buffer for which we are setting attributes

const struct attribute ** attrs Pointer to a null terminated list of pointers to attributes

bool **iio_validate_scan_mask_onehot**(struct iio_dev * indio_dev, const unsigned long * mask)
Validates that exactly one channel is selected

Parameters

struct iio_dev * indio_dev the iio device

const unsigned long * mask scan mask to be checked

Description

Return true if exactly one bit is set in the scan mask, false otherwise. It can be used for devices where only one channel can be active for sampling at a time.

int **iio_push_to_buffers**(struct iio_dev * indio_dev, const void * data)
push to a registered buffer.

Parameters

struct iio_dev * indio_dev iio_dev structure for device.

const void * data Full scan.

struct iio_buffer * **iio_buffer_get**(struct iio_buffer * buffer)
Grab a reference to the buffer

Parameters

struct iio_buffer * buffer The buffer to grab a reference for, may be NULL

Description

Returns the pointer to the buffer that was passed into the function.

void **iio_buffer_put**(struct iio_buffer * buffer)
Release the reference to the buffer

Parameters

struct iio_buffer * buffer The buffer to release the reference for, may be NULL

void **iio_device_attach_buffer**(struct iio_dev * indio_dev, struct iio_buffer * buffer)
Attach a buffer to a IIO device

Parameters

struct iio_dev * indio_dev The device the buffer should be attached to

struct iio_buffer * buffer The buffer to attach to the device

Description

This function attaches a buffer to a IIO device. The buffer stays attached to the device until the device is freed. The function should only be called at most once per device.

18.4 Triggers

- struct `iio_trigger` —industrial I/O trigger device
- `devm_iio_trigger_alloc()` —Resource-managed iio_trigger_alloc
- `devm_iio_trigger_register()` —Resource-managed iio_trigger_register iio_trigger_unregister
- `iio_trigger_validate_own_device()` —Check if a trigger and IIO device belong to the same device

In many situations it is useful for a driver to be able to capture data based on some external event (trigger) as opposed to periodically polling for data. An IIO trigger can be provided by a device driver that also has an IIO device based on hardware generated events (e.g. data ready or threshold exceeded) or provided by a separate driver from an independent interrupt source (e.g. GPIO line connected to some external system, timer interrupt or user space writing a specific file in sysfs). A trigger may initiate data capture for a number of sensors and also it may be completely unrelated to the sensor itself.

18.4.1 IIO trigger sysfs interface

There are two locations in sysfs related to triggers:

- `/sys/bus/iio/devices/triggerY/*`, this file is created once an IIO trigger is registered with the IIO core and corresponds to trigger with index Y. Because triggers can be very different depending on type there are few standard attributes that we can describe here:
  - `name`, trigger name that can be later used for association with a device.
  - `sampling_frequency`, some timer based triggers use this attribute to specify the frequency for trigger calls.
- `/sys/bus/iio/devices/iio:deviceX/trigger/*`, this directory is created once the device supports a triggered buffer. We can associate a trigger with our device by writing the trigger's name in the `current_trigger` file.

18.4.2 IIO trigger setup

Let's see a simple example of how to setup a trigger to be used by a driver:

```
struct iio_trigger_ops trigger_ops = {
    .set_trigger_state = sample_trigger_state,
    .validate_device = sample_validate_device,
}

struct iio_trigger *trig;

/* first, allocate memory for our trigger */
trig = iio_trigger_alloc(dev, "trig-%s-%d", name, idx);

/* setup trigger operations field */
trig->ops = &trigger_ops;

/* now register the trigger with the IIO core */
iio_trigger_register(trig);
```

18.4.3 IIO trigger ops

- struct `iio_trigger_ops` —operations structure for an iio_trigger.

Notice that a trigger has a set of operations attached:

- `set_trigger_state`, switch the trigger on/off on demand.
- `validate_device`, function to validate the device when the current trigger gets changed.

18.4.4 More details

struct **iio_trigger_ops**
operations structure for an iio_trigger.

Definition

```
struct iio_trigger_ops {
  int (*set_trigger_state)(struct iio_trigger *trig, bool state);
  int (*try_reenable)(struct iio_trigger *trig);
  int (*validate_device)(struct iio_trigger *trig, struct iio_dev *indio_
↪dev);
};
```

Members

set_trigger_state switch on/off the trigger on demand

try_reenable function to reenable the trigger when the use count is zero (may be NULL)

validate_device function to validate the device when the current trigger gets changed.

Description

This is typically static const within a driver and shared by instances of a given device.

struct **iio_trigger**
industrial I/O trigger device

Definition

```
struct iio_trigger {
  const struct iio_trigger_ops    *ops;
  struct module                   *owner;
  int id;
  const char                      *name;
  struct device                   dev;
  struct list_head                list;
  struct list_head                alloc_list;
  atomic_t use_count;
  struct irq_chip                 subirq_chip;
  int subirq_base;
  struct iio_subirq subirqs[CONFIG_IIO_CONSUMERS_PER_TRIGGER];
  unsigned long pool[BITS_TO_LONGS(CONFIG_IIO_CONSUMERS_PER_TRIGGER)];
  struct mutex                    pool_lock;
  bool attached_own_device;
};
```

Members

ops [DRIVER] operations structure

owner [INTERN] owner of this driver module

id [INTERN] unique id number

name [DRIVER] unique name

dev [DRIVER] associated device (if relevant)

list [INTERN] used in maintenance of global trigger list

alloc_list [DRIVER] used for driver specific trigger list

use_count [INTERN] use count for the trigger.

subirq_chip [INTERN] associate ‘virtual’ irq chip.

subirq_base [INTERN] base number for irqs provided by trigger.

subirqs [INTERN] information about the ‘child’ irqs.

pool [INTERN] bitmap of irqs currently in use.

pool_lock [INTERN] protection of the irq pool.

attached_own_device [INTERN] if we are using our own device as trigger, i.e. if we registered a poll function to the same device as the one providing the trigger.

void **iio_trigger_set_drvdata**(struct iio_trigger * trig, void * data)
Set trigger driver data

Parameters

struct iio_trigger * trig IIO trigger structure

void * data Driver specific data

Description

Allows to attach an arbitrary pointer to an IIO trigger, which can later be retrieved by `iio_trigger_get_drvdata()`.

void * **iio_trigger_get_drvdata**(struct iio_trigger * trig)
Get trigger driver data

Parameters

struct iio_trigger * trig IIO trigger structure

Description

Returns the data previously set with `iio_trigger_set_drvdata()`

iio_trigger_register(trig_info)
register a trigger with the IIO core

Parameters

trig_info trigger to be registered

void **iio_trigger_unregister**(struct iio_trigger * trig_info)
unregister a trigger from the core

Parameters

struct iio_trigger * trig_info trigger to be unregistered

int **iio_trigger_set_immutable**(struct iio_dev * indio_dev, struct iio_trigger * trig)
set an immutable trigger on destination

Parameters

struct iio_dev * indio_dev IIO device structure containing the device

struct iio_trigger * trig trigger to assign to device

void **iio_trigger_poll**(struct iio_trigger * trig)
called on a trigger occurring

Parameters

struct iio_trigger * trig trigger which occurred

Description

Typically called in relevant hardware interrupt handler.

bool **iio_trigger_using_own**(struct iio_dev * indio_dev)
tells us if we use our own HW trigger ourselves

Parameters

struct iio_dev * indio_dev device to check

struct iio_trigger * **devm_iio_trigger_alloc**(struct device * dev, const char * fmt, ...)
Resource-managed iio_trigger_alloc()

Parameters

struct device * dev Device to allocate iio_trigger for

const char * fmt trigger name format. If it includes format specifiers, the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

... variable arguments

Description

Managed iio_trigger_alloc. iio_trigger allocated with this function is automatically freed on driver detach.

Return

Pointer to allocated iio_trigger on success, NULL on failure.

int **__devm_iio_trigger_register**(struct device * dev, struct iio_trigger * trig_info, struct module * this_mod)
Resource-managed `iio_trigger_register()`

Parameters

struct device * dev device this trigger was allocated for

struct iio_trigger * trig_info trigger to register

struct module * this_mod module registering the trigger

Description

Managed `iio_trigger_register()`. The IIO trigger registered with this function is automatically unregistered on driver detach. This function calls `iio_trigger_register()` internally. Refer to that function for more information.

Return

0 on success, negative error number on failure.

int **iio_trigger_validate_own_device**(struct iio_trigger * trig, struct iio_dev * indio_dev)

Check if a trigger and IIO device belong to the same device

Parameters

struct iio_trigger * trig The IIO trigger to check

struct iio_dev * indio_dev the IIO device to check

Description

This function can be used as the validate_device callback for triggers that can only be attached to their own device.

Return

0 if both the trigger and the IIO device belong to the same device, -EINVAL otherwise.

18.5 Triggered Buffers

Now that we know what buffers and triggers are let's see how they work together.

18.5.1 IIO triggered buffer setup

- `iio_triggered_buffer_setup()` —Setup triggered buffer and pollfunc
- `iio_triggered_buffer_cleanup()` —Free resources allocated by `iio_triggered_buffer_setup()`
- struct `iio_buffer_setup_ops` —buffer setup related callbacks

A typical triggered buffer setup looks like this:

```
const struct iio_buffer_setup_ops sensor_buffer_setup_ops = {
  .preenable    = sensor_buffer_preenable,
  .postenable   = sensor_buffer_postenable,
  .postdisable  = sensor_buffer_postdisable,
  .predisable   = sensor_buffer_predisable,
};

irqreturn_t sensor_iio_pollfunc(int irq, void *p)
{
    pf->timestamp = iio_get_time_ns((struct indio_dev *)p);
    return IRQ_WAKE_THREAD;
}

irqreturn_t sensor_trigger_handler(int irq, void *p)
{
    u16 buf[8];
    int i = 0;
```

(continues on next page)

(continued from previous page)

```
    /* read data for each active channel */
    for_each_set_bit(bit, active_scan_mask, masklength)
        buf[i++] = sensor_get_data(bit)

    iio_push_to_buffers_with_timestamp(indio_dev, buf, timestamp);

    iio_trigger_notify_done(trigger);
    return IRQ_HANDLED;
}

/* setup triggered buffer, usually in probe function */
iio_triggered_buffer_setup(indio_dev, sensor_iio_polfunc,
                           sensor_trigger_handler,
                           sensor_buffer_setup_ops);
```

The important things to notice here are:

- `iio_buffer_setup_ops`, the buffer setup functions to be called at predefined points in the buffer configuration sequence (e.g. before enable, after disable). If not specified, the IIO core uses the default iio_triggered_buffer_setup_ops.
- **sensor_iio_pollfunc**, the function that will be used as top half of poll function. It should do as little processing as possible, because it runs in interrupt context. The most common operation is recording of the current timestamp and for this reason one can use the IIO core defined `iio_pollfunc_store_time()` function.
- **sensor_trigger_handler**, the function that will be used as bottom half of the poll function. This runs in the context of a kernel thread and all the processing takes place here. It usually reads data from the device and stores it in the internal buffer together with the timestamp recorded in the top half.

18.5.2 More details

int **iio_triggered_buffer_setup**(struct iio_dev * indio_dev, irqreturn_t (*h)(int irq, void *p), irqreturn_t (*thread) (int irq, void *p), const struct iio_buffer_setup_ops * setup_ops)

Setup triggered buffer and pollfunc

Parameters

struct iio_dev * indio_dev IIO device structure

irqreturn_t (*)(int irq, void *p) h Function which will be used as pollfunc top half

irqreturn_t (*)(int irq, void *p) thread Function which will be used as pollfunc bottom half

const struct iio_buffer_setup_ops * setup_ops Buffer setup functions to use for this device. If NULL the default setup functions for triggered buffers will be used.

Description

This function combines some common tasks which will normally be performed when setting up a triggered buffer. It will allocate the buffer and the pollfunc.

Before calling this function the indio_dev structure should already be completely initialized, but not yet registered. In practice this means that this function should be called right before `iio_device_register()`.

To free the resources allocated by this function call `iio_triggered_buffer_cleanup()`.

void **iio_triggered_buffer_cleanup**(struct iio_dev * indio_dev)
Free resources allocated by `iio_triggered_buffer_setup()`

Parameters

struct iio_dev * indio_dev IIO device structure

18.6 HW consumer

An IIO device can be directly connected to another device in hardware. In this case the buffers between IIO provider and IIO consumer are handled by hardware. The Industrial I/O HW consumer offers a way to bond these IIO devices without software buffer for data. The implementation can be found under `drivers/iio/buffer/hw-consumer.c`

- struct `iio_hw_consumer` —Hardware consumer structure
- `iio_hw_consumer_alloc()` —Allocate IIO hardware consumer
- `iio_hw_consumer_free()` —Free IIO hardware consumer
- `iio_hw_consumer_enable()` —Enable IIO hardware consumer
- `iio_hw_consumer_disable()` —Disable IIO hardware consumer

18.6.1 HW consumer setup

As standard IIO device the implementation is based on IIO provider/consumer. A typical IIO HW consumer setup looks like this:

```
static struct iio_hw_consumer *hwc;

static const struct iio_info adc_info = {
        .read_raw = adc_read_raw,
};

static int adc_read_raw(struct iio_dev *indio_dev,
                        struct iio_chan_spec const *chan, int *val,
                        int *val2, long mask)
{
        ret = iio_hw_consumer_enable(hwc);

        /* Acquire data */

        ret = iio_hw_consumer_disable(hwc);
```

(continues on next page)

(continued from previous page)

```
}

static int adc_probe(struct platform_device *pdev)
{
        hwc = devm_iio_hw_consumer_alloc(&iio->dev);
}
```

18.6.2 More details

struct iio_hw_consumer * **iio_hw_consumer_alloc**(struct device * dev)
Allocate IIO hardware consumer

Parameters

struct device * dev Pointer to consumer device.

Description

Returns a valid iio_hw_consumer on success or a ERR_PTR() on failure.

void **iio_hw_consumer_free**(struct iio_hw_consumer * hwc)
Free IIO hardware consumer

Parameters

struct iio_hw_consumer * hwc hw consumer to free.

struct iio_hw_consumer * **devm_iio_hw_consumer_alloc**(struct device * dev)
Resource-managed `iio_hw_consumer_alloc()`

Parameters

struct device * dev Pointer to consumer device.

Description

Managed iio_hw_consumer_alloc. iio_hw_consumer allocated with this function is automatically freed on driver detach.

returns pointer to allocated iio_hw_consumer on success, NULL on failure.

int **iio_hw_consumer_enable**(struct iio_hw_consumer * hwc)
Enable IIO hardware consumer

Parameters

struct iio_hw_consumer * hwc iio_hw_consumer to enable.

Description

Returns 0 on success.

void **iio_hw_consumer_disable**(struct iio_hw_consumer * hwc)
Disable IIO hardware consumer

Parameters

struct iio_hw_consumer * hwc iio_hw_consumer to disable.

CHAPTER

NINETEEN

INPUT SUBSYSTEM

19.1 Input core

struct **input_value**
input value representation

Definition

```
struct input_value {
  __u16 type;
  __u16 code;
  __s32 value;
};
```

Members

type type of value (EV_KEY, EV_ABS, etc)

code the value code

value the value

struct **input_dev**
represents an input device

Definition

```
struct input_dev {
  const char *name;
  const char *phys;
  const char *uniq;
  struct input_id id;
  unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
  unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
  unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
  unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
  unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
  unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
  unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
  unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
  unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
  unsigned long swbit[BITS_TO_LONGS(SW_CNT)];
  unsigned int hint_events_per_packet;
  unsigned int keycodemax;
  unsigned int keycodesize;
```

(continues on next page)

(continued from previous page)

```
  void *keycode;
  int (*setkeycode)(struct input_dev *dev,const struct input_keymap_entry␣
↪*ke, unsigned int *old_keycode);
  int (*getkeycode)(struct input_dev *dev, struct input_keymap_entry *ke);
  struct ff_device *ff;
  struct input_dev_poller *poller;
  unsigned int repeat_key;
  struct timer_list timer;
  int rep[REP_CNT];
  struct input_mt *mt;
  struct input_absinfo *absinfo;
  unsigned long key[BITS_TO_LONGS(KEY_CNT)];
  unsigned long led[BITS_TO_LONGS(LED_CNT)];
  unsigned long snd[BITS_TO_LONGS(SND_CNT)];
  unsigned long sw[BITS_TO_LONGS(SW_CNT)];
  int (*open)(struct input_dev *dev);
  void (*close)(struct input_dev *dev);
  int (*flush)(struct input_dev *dev, struct file *file);
  int (*event)(struct input_dev *dev, unsigned int type, unsigned int code,
↪ int value);
  struct input_handle __rcu *grab;
  spinlock_t event_lock;
  struct mutex mutex;
  unsigned int users;
  bool going_away;
  struct device dev;
  struct list_head        h_list;
  struct list_head        node;
  unsigned int num_vals;
  unsigned int max_vals;
  struct input_value *vals;
  bool devres_managed;
  ktime_t timestamp[INPUT_CLK_MAX];
};
```

Members

name name of the device

phys physical path to the device in the system hierarchy

uniq unique identification code for the device (if device has it)

id id of the device (struct input_id)

propbit bitmap of device properties and quirks

evbit bitmap of types of events supported by the device (EV_KEY, EV_REL, etc.)

keybit bitmap of keys/buttons this device has

relbit bitmap of relative axes for the device

absbit bitmap of absolute axes for the device

mscbit bitmap of miscellaneous events supported by the device

ledbit bitmap of leds present on the device

sndbit bitmap of sound effects supported by the device

ffbit bitmap of force feedback effects supported by the device

swbit bitmap of switches present on the device

hint_events_per_packet average number of events generated by the device in a packet (between EV_SYN/SYN_REPORT events). Used by event handlers to estimate size of the buffer needed to hold events.

keycodemax size of keycode table

keycodesize size of elements in keycode table

keycode map of scancodes to keycodes for this device

setkeycode optional method to alter current keymap, used to implement sparse keymaps. If not supplied default mechanism will be used. The method is being called while holding event_lock and thus must not sleep

getkeycode optional legacy method to retrieve current keymap.

ff force feedback structure associated with the device if device supports force feedback effects

poller poller structure associated with the device if device is set up to use polling mode

repeat_key stores key code of the last key pressed; used to implement software autorepeat

timer timer for software autorepeat

rep current values for autorepeat parameters (delay, rate)

mt pointer to multitouch state

absinfo array of `struct input_absinfo` elements holding information about absolute axes (current value, min, max, flat, fuzz, resolution)

key reflects current state of device's keys/buttons

led reflects current state of device's LEDs

snd reflects current state of sound effects

sw reflects current state of device's switches

open this method is called when the very first user calls `input_open_device()`. The driver must prepare the device to start generating events (start polling thread, request an IRQ, submit URB, etc.)

close this method is called when the very last user calls `input_close_device()`.

flush purges the device. Most commonly used to get rid of force feedback effects loaded into the device when disconnecting from it

event event handler for events sent _to_ the device, like EV_LED or EV_SND. The device is expected to carry out the requested action (turn on a LED, play sound, etc.) The call is protected by **event_lock** and must not sleep

grab input handle that currently has the device grabbed (via EVIOCGRAB ioctl). When a handle grabs a device it becomes sole recipient for all input events coming from the device

event_lock this spinlock is taken when input core receives and processes a new event for the device (in `input_event()`). Code that accesses and/or modifies parameters of a device (such as keymap or absmin, absmax, absfuzz, etc.) after device has been registered with input core must take this lock.

mutex serializes calls to open(), close() and flush() methods

users stores number of users (input handlers) that opened this device. It is used by `input_open_device()` and `input_close_device()` to make sure that dev->open() is only called when the first user opens device and dev->close() is called when the very last user closes the device

going_away marks devices that are in a middle of unregistering and causes input_open_device*() fail with -ENODEV.

dev driver model' s view of this device

h_list list of input handles associated with the device. When accessing the list dev->mutex must be held

node used to place the device onto input_dev_list

num_vals number of values queued in the current frame

max_vals maximum number of values queued in a frame

vals array of values queued in the current frame

devres_managed indicates that devices is managed with devres framework and needs not be explicitly unregistered or freed.

timestamp storage for a timestamp set by input_set_timestamp called by a driver

struct **input_handler**
implements one of interfaces for input devices

Definition

```
struct input_handler {
  void *private;
  void (*event)(struct input_handle *handle, unsigned int type, unsigned␣
↪int code, int value);
  void (*events)(struct input_handle *handle, const struct input_value␣
↪*vals, unsigned int count);
  bool (*filter)(struct input_handle *handle, unsigned int type, unsigned␣
↪int code, int value);
  bool (*match)(struct input_handler *handler, struct input_dev *dev);
  int (*connect)(struct input_handler *handler, struct input_dev *dev,␣
↪const struct input_device_id *id);
  void (*disconnect)(struct input_handle *handle);
  void (*start)(struct input_handle *handle);
  bool legacy_minors;
  int minor;
  const char *name;
  const struct input_device_id *id_table;
  struct list_head        h_list;
  struct list_head        node;
};
```

Members

private driver-specific data

event event handler. This method is being called by input core with interrupts disabled and dev->event_lock spinlock held and so it may not sleep

events event sequence handler. This method is being called by input core with interrupts disabled and dev->event_lock spinlock held and so it may not sleep

filter similar to **event**; separates normal event handlers from "filters".

match called after comparing device's id with handler's id_table to perform fine-grained matching between device and handler

connect called when attaching a handler to an input device

disconnect disconnects a handler from input device

start starts handler for given handle. This function is called by input core right after connect() method and also when a process that "grabbed" a device releases it

legacy_minors set to `true` by drivers using legacy minor ranges

minor beginning of range of 32 legacy minors for devices this driver can provide

name name of the handler, to be shown in /proc/bus/input/handlers

id_table pointer to a table of input_device_ids this driver can handle

h_list list of input handles associated with the handler

node for placing the driver onto input_handler_list

Description

Input handlers attach to input devices and create input handles. There are likely several handlers attached to any given input device at the same time. All of them will get their copy of input event generated by the device.

The very same structure is used to implement input filters. Input core allows filters to run first and will not pass event to regular handlers if any of the filters indicate that the event should be filtered (by returning `true` from their filter() method).

Note that input core serializes calls to connect() and disconnect() methods.

struct **input_handle**
links input device with an input handler

Definition

```
struct input_handle {
  void *private;
  int open;
  const char *name;
  struct input_dev *dev;
  struct input_handler *handler;
  struct list_head        d_node;
  struct list_head        h_node;
};
```

Members

private handler-specific data

open counter showing whether the handle is ‘open’, i.e. should deliver events from its device

name name given to the handle by handler that created it

dev input device the handle is attached to

handler handler that works with the device through this handle

d_node used to put the handle on device’s list of attached handles

h_node used to put the handle on handler’s list of handles from which it gets events

void **input_set_events_per_packet**(struct input_dev * dev, int n_events)
tell handlers about the driver event rate

Parameters

struct input_dev * dev the input device used by the driver

int n_events the average number of events between calls to input_sync()

Description

If the event rate sent from a device is unusually large, use this function to set the expected event rate. This will allow handlers to set up an appropriate buffer size for the event stream, in order to minimize information loss.

struct **ff_device**
force-feedback part of an input device

Definition

```
struct ff_device {
  int (*upload)(struct input_dev *dev, struct ff_effect *effect, struct ff_
↪effect *old);
  int (*erase)(struct input_dev *dev, int effect_id);
  int (*playback)(struct input_dev *dev, int effect_id, int value);
  void (*set_gain)(struct input_dev *dev, u16 gain);
  void (*set_autocenter)(struct input_dev *dev, u16 magnitude);
  void (*destroy)(struct ff_device *);
  void *private;
  unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
  struct mutex mutex;
  int max_effects;
  struct ff_effect *effects;
  struct file *effect_owners[];
};
```

Members

upload Called to upload an new effect into device

erase Called to erase an effect from device

playback Called to request device to start playing specified effect

set_gain Called to set specified gain

set_autocenter Called to auto-center device

destroy called by input core when parent input device is being destroyed

private driver-specific data, will be freed automatically

ffbit bitmap of force feedback capabilities truly supported by device (not emulated like ones in input_dev->ffbit)

mutex mutex for serializing access to the device

max_effects maximum number of effects supported by device

effects pointer to an array of effects currently loaded into device

effect_owners array of effect owners; when file handle owning an effect gets closed the effect is automatically erased

Description

Every force-feedback device must implement upload() and playback() methods; erase() is optional. set_gain() and set_autocenter() need only be implemented if driver sets up FF_GAIN and FF_AUTOCENTER bits.

Note that playback(), set_gain() and set_autocenter() are called with dev->event_lock spinlock held and interrupts off and thus may not sleep.

void **input_event**(struct input_dev * dev, unsigned int type, unsigned int code, int value)
report new input event

Parameters

struct input_dev * dev device that generated the event

unsigned int type type of the event

unsigned int code event code

int value value of the event

Description

This function should be used by drivers implementing various input devices to report input events. See also `input_inject_event()`.

NOTE

`input_event()` may be safely used right after input device was allocated with `input_allocate_device()`, even before it is registered with `input_register_device()`, but the event will not reach any of the input handlers. Such early invocation of `input_event()` may be used to 'seed' initial state of a switch or initial position of absolute axis, etc.

void **input_inject_event**(struct input_handle * handle, unsigned int type, unsigned int code, int value)
send input event from input handler

Parameters

struct input_handle * handle input handle to send event through

unsigned int type type of the event

unsigned int code event code

int value value of the event

Description

Similar to `input_event()` but will ignore event if device is “grabbed” and handle injecting event is not the one that owns the device.

void **input_alloc_absinfo**(struct input_dev * dev)
allocates array of input_absinfo structs

Parameters

struct input_dev * dev the input device emitting absolute events

Description

If the absinfo struct the caller asked for is already allocated, this functions will not do anything.

int **input_grab_device**(struct input_handle * handle)
grabs device for exclusive use

Parameters

struct input_handle * handle input handle that wants to own the device

Description

When a device is grabbed by an input handle all events generated by the device are delivered only to this handle. Also events injected by other input handles are ignored while device is grabbed.

void **input_release_device**(struct input_handle * handle)
release previously grabbed device

Parameters

struct input_handle * handle input handle that owns the device

Description

Releases previously grabbed device so that other input handles can start receiving input events. Upon release all handlers attached to the device have their start() method called so they have a change to synchronize device state with the rest of the system.

int **input_open_device**(struct input_handle * handle)
open input device

Parameters

struct input_handle * handle handle through which device is being accessed

Description

This function should be called by input handlers when they want to start receive events from given input device.

void **input_close_device**(struct input_handle * handle)
close input device

Parameters

struct input_handle * handle handle through which device is being accessed

Description

This function should be called by input handlers when they want to stop receive events from given input device.

int **input_scancode_to_scalar**(const struct input_keymap_entry * ke, unsigned int * scancode)
converts scancode in `struct input_keymap_entry`

Parameters

const struct input_keymap_entry * ke keymap entry containing scancode to be converted.

unsigned int * scancode pointer to the location where converted scancode should be stored.

Description

This function is used to convert scancode stored in `struct keymap_entry` into scalar form understood by legacy keymap handling methods. These methods expect scancodes to be represented as 'unsigned int'.

int **input_get_keycode**(struct input_dev * dev, struct input_keymap_entry * ke)
retrieve keycode currently mapped to a given scancode

Parameters

struct input_dev * dev input device which keymap is being queried

struct input_keymap_entry * ke keymap entry

Description

This function should be called by anyone interested in retrieving current keymap. Presently evdev handlers use it.

int **input_set_keycode**(struct input_dev * dev, const struct input_keymap_entry * ke)
attribute a keycode to a given scancode

Parameters

struct input_dev * dev input device which keymap is being updated

const struct input_keymap_entry * ke new keymap entry

Description

This function should be called by anyone needing to update current keymap. Presently keyboard and evdev handlers use it.

void **input_reset_device**(struct input_dev * dev)
reset/restore the state of input device

Parameters

struct input_dev * dev input device whose state needs to be reset

Description

This function tries to reset the state of an opened input device and bring internal state and state if the hardware in sync with each other. We mark all keys as released, restore LED state, repeat rate, etc.

struct input_dev * **input_allocate_device**(void)
allocate memory for new input device

Parameters

void no arguments

Description

Returns prepared struct input_dev or `NULL`.

NOTE

Use `input_free_device()` to free devices that have not been registered; `input_unregister_device()` should be used for already registered devices.

struct input_dev * **devm_input_allocate_device**(struct device * dev)
allocate managed input device

Parameters

struct device * dev device owning the input device being created

Description

Returns prepared struct input_dev or `NULL`.

Managed input devices do not need to be explicitly unregistered or freed as it will be done automatically when owner device unbinds from its driver (or binding fails). Once managed input device is allocated, it is ready to be set up and registered in the same fashion as regular input device. There are no special devm_input_device_[un]register() variants, regular ones work with both managed and unmanaged devices, should you need them. In most cases however, managed input device need not be explicitly unregistered or freed.

NOTE

the owner device is set up as parent of input device and users should not override it.

void **input_free_device**(struct input_dev * dev)
free memory occupied by input_dev structure

Parameters

struct input_dev * dev input device to free

Description

This function should only be used if `input_register_device()` was not called yet or if it failed. Once device was registered use `input_unregister_device()` and memory will be freed once last reference to the device is dropped.

Device should be allocated by `input_allocate_device()`.

NOTE

If there are references to the input device then memory will not be freed until last reference is dropped.

void **input_set_timestamp**(struct input_dev * dev, ktime_t timestamp)
set timestamp for input events

Parameters

struct input_dev * dev input device to set timestamp for

ktime_t timestamp the time at which the event has occurred in CLOCK_MONOTONIC

Description

This function is intended to provide to the input system a more accurate time of when an event actually occurred. The driver should call this function as soon as a timestamp is acquired ensuring clock conversions in input_set_timestamp are done correctly.

The system entering suspend state between timestamp acquisition and calling input_set_timestamp can result in inaccurate conversions.

ktime_t * **input_get_timestamp**(struct input_dev * dev)
get timestamp for input events

Parameters

struct input_dev * dev input device to get timestamp from

Description

A valid timestamp is a timestamp of non-zero value.

void **input_set_capability**(struct input_dev * dev, unsigned int type, unsigned int code)
mark device as capable of a certain event

Parameters

struct input_dev * dev device that is capable of emitting or accepting event

unsigned int type type of the event (EV_KEY, EV_REL, etc…)

unsigned int code event code

Description

In addition to setting up corresponding bit in appropriate capability bitmap the function also adjusts dev->evbit.

void **input_enable_softrepeat**(struct input_dev * dev, int delay, int period)
enable software autorepeat

Parameters

struct input_dev * dev input device

int delay repeat delay

int period repeat period

Description

Enable software autorepeat on the input device.

int **input_register_device**(struct input_dev * dev)
register device with input core

Parameters

struct input_dev * dev device to be registered

Description

This function registers device with input core. The device must be allocated with `input_allocate_device()` and all it' s capabilities set up before registering. If function fails the device must be freed with `input_free_device()`. Once device has been successfully registered it can be unregistered with `input_unregister_device()`; `input_free_device()` should not be called in this case.

Note that this function is also used to register managed input devices (ones allocated with `devm_input_allocate_device()`). Such managed input devices need not be explicitly unregistered or freed, their tear down is controlled by the devres infrastructure. It is also worth noting that tear down of managed input devices is internally a 2-step process: registered managed input device is first unregistered, but stays in memory and can still handle `input_event()` calls (although events will not be delivered anywhere). The freeing of managed input device will happen later, when devres stack is unwound to the point where device allocation was made.

void **input_unregister_device**(struct input_dev * dev)
unregister previously registered device

Parameters

struct input_dev * dev device to be unregistered

Description

This function unregisters an input device. Once device is unregistered the caller should not try to access it as it may get freed at any moment.

int **input_register_handler**(struct input_handler * handler)
register a new input handler

Parameters

struct input_handler * handler handler to be registered

Description

This function registers a new input handler (interface) for input devices in the system and attaches it to all input devices that are compatible with the handler.

void **input_unregister_handler**(struct input_handler * handler)
unregisters an input handler

Parameters

struct input_handler * handler handler to be unregistered

Description

This function disconnects a handler from its input devices and removes it from lists of known handlers.

int **input_handler_for_each_handle**(struct input_handler * handler, void * data, int (*fn)(struct input_handle *, void *))
handle iterator

Parameters

struct input_handler * handler input handler to iterate

void * data data for the callback

int (*)(struct input_handle *, void *) fn function to be called for each handle

Description

Iterate over **bus**'s list of devices, and call **fn** for each, passing it **data** and stop when **fn** returns a non-zero value. The function is using RCU to traverse the list and therefore may be using in atomic contexts. The **fn** callback is invoked from RCU critical section and thus must not sleep.

int **input_register_handle**(struct input_handle * handle)
register a new input handle

Parameters

struct input_handle * handle handle to register

Description

This function puts a new input handle onto device's and handler's lists so that events can flow through it once it is opened using `input_open_device()`.

This function is supposed to be called from handler's connect() method.

void **input_unregister_handle**(struct input_handle * handle)
unregister an input handle

Parameters

struct input_handle * handle handle to unregister

Description

This function removes input handle from device's and handler's lists.

This function is supposed to be called from handler's disconnect() method.

int **input_get_new_minor**(int legacy_base, unsigned int legacy_num, bool allow_dynamic)
allocates a new input minor number

Parameters

int legacy_base beginning or the legacy range to be searched

unsigned int legacy_num size of legacy range

bool allow_dynamic whether we can also take ID from the dynamic range

Description

This function allocates a new device minor for from input major namespace. Caller can request legacy minor by specifying **legacy_base** and **legacy_num** parameters

and whether ID can be allocated from dynamic range if there are no free IDs in legacy range.

void **input_free_minor**(unsigned int minor)
release previously allocated minor

Parameters

unsigned int minor minor to be released

Description

This function releases previously allocated input minor so that it can be reused later.

int **input_ff_upload**(struct input_dev * dev, struct ff_effect * effect, struct file * file)
upload effect into force-feedback device

Parameters

struct input_dev * dev input device

struct ff_effect * effect effect to be uploaded

struct file * file owner of the effect

int **input_ff_erase**(struct input_dev * dev, int effect_id, struct file * file)
erase a force-feedback effect from device

Parameters

struct input_dev * dev input device to erase effect from

int effect_id id of the effect to be erased

struct file * file purported owner of the request

Description

This function erases a force-feedback effect from specified device. The effect will only be erased if it was uploaded through the same file handle that is requesting erase.

int **input_ff_event**(struct input_dev * dev, unsigned int type, unsigned int code, int value)
generic handler for force-feedback events

Parameters

struct input_dev * dev input device to send the effect to

unsigned int type event type (anything but EV_FF is ignored)

unsigned int code event code

int value event value

int **input_ff_create**(struct input_dev * dev, unsigned int max_effects)
create force-feedback device

Parameters

struct input_dev * dev input device supporting force-feedback

unsigned int max_effects maximum number of effects supported by the device

Description

This function allocates all necessary memory for a force feedback portion of an input device and installs all default handlers. **dev->ffbit** should be already set up before calling this function. Once ff device is created you need to setup its upload, erase, playback and other handlers before registering input device

void **input_ff_destroy**(struct input_dev * dev)
frees force feedback portion of input device

Parameters

struct input_dev * dev input device supporting force feedback

Description

This function is only needed in error path as input core will automatically free force feedback structures when device is destroyed.

int **input_ff_create_memless**(struct input_dev * dev, void * data, int (*play_effect)(struct input_dev *, void *, struct ff_effect *))
create memoryless force-feedback device

Parameters

struct input_dev * dev input device supporting force-feedback

void * data driver-specific data to be passed into **play_effect**

int (*)(struct input_dev *, void *, struct ff_effect *) play_effect
driver-specific method for playing FF effect

19.2 Multitouch Library

struct **input_mt_slot**
represents the state of an input MT slot

Definition

```
struct input_mt_slot {
  int abs[ABS_MT_LAST - ABS_MT_FIRST + 1];
  unsigned int frame;
  unsigned int key;
};
```

Members

abs holds current values of ABS_MT axes for this slot

frame last frame at which `input_mt_report_slot_state()` was called

key optional driver designation of this slot

struct **input_mt**
state of tracked contacts

Definition

```
struct input_mt {
  int trkid;
  int num_slots;
  int slot;
  unsigned int flags;
  unsigned int frame;
  int *red;
  struct input_mt_slot slots[];
};
```

Members

trkid stores MT tracking ID for the next contact

num_slots number of MT slots the device uses

slot MT slot currently being transmitted

flags input_mt operation flags

frame increases every time `input_mt_sync_frame()` is called

red reduced cost matrix for in-kernel tracking

slots array of slots holding current values of tracked contacts

struct **input_mt_pos**
    contact position

Definition

```
struct input_mt_pos {
  s16 x, y;
};
```

Members

x horizontal coordinate

y vertical coordinate

int **input_mt_init_slots**(struct input_dev * dev, unsigned int num_slots,
unsigned int flags)
    initialize MT input slots

Parameters

struct input_dev * dev input device supporting MT events and finger tracking

unsigned int num_slots number of slots used by the device

unsigned int flags mt tasks to handle in core

Description

This function allocates all necessary memory for MT slot handling in the input device, prepares the ABS_MT_SLOT and ABS_MT_TRACKING_ID events for use and sets up appropriate buffers. Depending on the flags set, it also performs pointer emulation and frame synchronization.

May be called repeatedly. Returns -EINVAL if attempting to reinitialize with a different number of slots.

void **input_mt_destroy_slots**(struct input_dev * dev)
frees the MT slots of the input device

Parameters

struct input_dev * dev input device with allocated MT slots

Description

This function is only needed in error path as the input core will automatically free the MT slots when the device is destroyed.

bool **input_mt_report_slot_state**(struct input_dev * dev, unsigned int tool_type, bool active)
report contact state

Parameters

struct input_dev * dev input device with allocated MT slots

unsigned int tool_type the tool type to use in this slot

bool active true if contact is active, false otherwise

Description

Reports a contact via ABS_MT_TRACKING_ID, and optionally ABS_MT_TOOL_TYPE. If active is true and the slot is currently inactive, or if the tool type is changed, a new tracking id is assigned to the slot. The tool type is only reported if the corresponding absbit field is set.

Returns true if contact is active.

void **input_mt_report_finger_count**(struct input_dev * dev, int count)
report contact count

Parameters

struct input_dev * dev input device with allocated MT slots

int count the number of contacts

Description

Reports the contact count via BTN_TOOL_FINGER, BTN_TOOL_DOUBLETAP, BTN_TOOL_TRIPLETAP and BTN_TOOL_QUADTAP.

The input core ensures only the KEY events already setup for this device will produce output.

void **input_mt_report_pointer_emulation**(struct input_dev * dev, bool use_count)
common pointer emulation

Parameters

struct input_dev * dev input device with allocated MT slots

bool use_count report number of active contacts as finger count

Description

Performs legacy pointer emulation via BTN_TOUCH, ABS_X, ABS_Y and ABS_PRESSURE. Touchpad finger count is emulated if use_count is true.

The input core ensures only the KEY and ABS axes already setup for this device will produce output.

void **input_mt_drop_unused**(struct input_dev * dev)
Inactivate slots not seen in this frame

Parameters

struct input_dev * dev input device with allocated MT slots

Description

Lift all slots not seen since the last call to this function.

void **input_mt_sync_frame**(struct input_dev * dev)
synchronize mt frame

Parameters

struct input_dev * dev input device with allocated MT slots

Description

Close the frame and prepare the internal state for a new one. Depending on the flags, marks unused slots as inactive and performs pointer emulation.

int **input_mt_assign_slots**(struct input_dev * dev, int * slots, const struct input_mt_pos * pos, int num_pos, int dmax)
perform a best-match assignment

Parameters

struct input_dev * dev input device with allocated MT slots

int * slots the slot assignment to be filled

const struct input_mt_pos * pos the position array to match

int num_pos number of positions

int dmax maximum ABS_MT_POSITION displacement (zero for infinite)

Description

Performs a best match against the current contacts and returns the slot assignment list. New contacts are assigned to unused slots.

The assignments are balanced so that all coordinate displacements are below the euclidian distance dmax. If no such assignment can be found, some contacts are assigned to unused slots.

Returns zero on success, or negative error in case of failure.

int **input_mt_get_slot_by_key**(struct input_dev * dev, int key)
return slot matching key

Parameters

struct input_dev * dev input device with allocated MT slots

int key the key of the sought slot

Description

Returns the slot of the given key, if it exists, otherwise set the key on the first unused slot and return.

If no available slot can be found, -1 is returned. Note that for this function to work properly, `input_mt_sync_frame()` has to be called at each frame.

19.3 Polled input devices

struct **input_polled_dev**
simple polled input device

Definition

```
struct input_polled_dev {
  void *private;
  void (*open)(struct input_polled_dev *dev);
  void (*close)(struct input_polled_dev *dev);
  void (*poll)(struct input_polled_dev *dev);
  unsigned int poll_interval;
  unsigned int poll_interval_max;
  unsigned int poll_interval_min;
  struct input_dev *input;
};
```

Members

private private driver data.

open driver-supplied method that prepares device for polling (enabled the device and maybe flushes device state).

close driver-supplied method that is called when device is no longer being polled. Used to put device into low power mode.

poll driver-supplied method that polls the device and posts input events (mandatory).

poll_interval specifies how often the poll() method should be called. Defaults to 500 msec unless overridden when registering the device.

poll_interval_max specifies upper bound for the poll interval. Defaults to the initial value of **poll_interval**.

poll_interval_min specifies lower bound for the poll interval. Defaults to 0.

input input device structure associated with the polled device. Must be properly initialized by the driver (id, name, phys, bits).

Description

Polled input device provides a skeleton for supporting simple input devices that do not raise interrupts but have to be periodically scanned or polled to detect changes in their state.

struct input_polled_dev * **input_allocate_polled_device**(void)
allocate memory for polled device

Parameters

void no arguments

Description

The function allocates memory for a polled device and also for an input device associated with this polled device.

struct input_polled_dev * **devm_input_allocate_polled_device**(struct device * dev)
allocate managed polled device

Parameters

struct device * dev device owning the polled device being created

Description

Returns prepared `struct input_polled_dev` or `NULL`.

Managed polled input devices do not need to be explicitly unregistered or freed as it will be done automatically when owner device unbinds from * its driver (or binding fails). Once such managed polled device is allocated, it is ready to be set up and registered in the same fashion as regular polled input devices (using `input_register_polled_device()` function).

If you want to manually unregister and free such managed polled devices, it can be still done by calling `input_unregister_polled_device()` and `input_free_polled_device()`, although it is rarely needed.

NOTE

the owner device is set up as parent of input device and users should not override it.

void **input_free_polled_device**(struct input_polled_dev * dev)
free memory allocated for polled device

Parameters

struct input_polled_dev * dev device to free

Description

The function frees memory allocated for polling device and drops reference to the associated input device.

int **input_register_polled_device**(struct input_polled_dev * dev)
register polled device

Parameters

struct input_polled_dev * dev device to register

Description

The function registers previously initialized polled input device with input layer. The device should be allocated with call to `input_allocate_polled_device()`. Callers should also set up poll() method and set up capabilities (id, name, phys, bits) of the corresponding input_dev structure.

void **input_unregister_polled_device**(struct input_polled_dev * dev)
unregister polled device

Parameters

struct input_polled_dev * dev device to unregister

Description

The function unregisters previously registered polled input device from input layer. Polling is stopped and device is ready to be freed with call to `input_free_polled_device()`.

19.4 Matrix keyboards/keypads

struct **matrix_keymap_data**
keymap for matrix keyboards

Definition

```
struct matrix_keymap_data {
  const uint32_t *keymap;
  unsigned int    keymap_size;
};
```

Members

keymap pointer to array of uint32 values encoded with KEY() macro representing keymap

keymap_size number of entries (initialized) in this keymap

Description

This structure is supposed to be used by platform code to supply keymaps to drivers that implement matrix-like keypads/keyboards.

struct **matrix_keypad_platform_data**
platform-dependent keypad data

Definition

```
struct matrix_keypad_platform_data {
  const struct matrix_keymap_data *keymap_data;
  const unsigned int *row_gpios;
  const unsigned int *col_gpios;
  unsigned int    num_row_gpios;
  unsigned int    num_col_gpios;
  unsigned int    col_scan_delay_us;
  unsigned int    debounce_ms;
  unsigned int    clustered_irq;
  unsigned int    clustered_irq_flags;
  bool active_low;
  bool wakeup;
  bool no_autorepeat;
  bool drive_inactive_cols;
};
```

Members

keymap_data pointer to `matrix_keymap_data`

row_gpios pointer to array of gpio numbers representing rows

col_gpios pointer to array of gpio numbers reporesenting colums

num_row_gpios actual number of row gpios used by device

num_col_gpios actual number of col gpios used by device

col_scan_delay_us delay, measured in microseconds, that is needed before we can keypad after activating column gpio

debounce_ms debounce interval in milliseconds

clustered_irq may be specified if interrupts of all row/column GPIOs are bundled to one single irq

clustered_irq_flags flags that are needed for the clustered irq

active_low gpio polarity

wakeup controls whether the device should be set up as wakeup source

no_autorepeat disable key autorepeat

drive_inactive_cols drive inactive columns during scan, rather than making them inputs.

Description

This structure represents platform-specific data that use used by matrix_keypad driver to perform proper initialization.

19.5 Sparse keymap support

struct **key_entry**
keymap entry for use in sparse keymap

Definition

```
struct key_entry {
  int type;
  u32 code;
  union {
    u16 keycode;
    struct {
      u8 code;
      u8 value;
    } sw;
  };
};
```

Members

type Type of the key entry (KE_KEY, KE_SW, KE_VSW, KE_END); drivers are allowed to extend the list with their own private definitions.

code Device-specific data identifying the button/switch

{unnamed_union} anonymous

keycode KEY_* code assigned to a key/button

sw.code SW_* code assigned to a switch

sw.value Value that should be sent in an input even when KE_SW switch is toggled. KE_VSW switches ignore this field and expect driver to supply value for the event.

Description

This structure defines an entry in a sparse keymap used by some input devices for which traditional table-based approach is not suitable.

struct key_entry * **sparse_keymap_entry_from_scancode**(struct input_dev * dev, unsigned int code)

perform sparse keymap lookup

Parameters

struct input_dev * dev Input device using sparse keymap

unsigned int code Scan code

Description

This function is used to perform `struct key_entry` lookup in an input device using sparse keymap.

struct key_entry * **sparse_keymap_entry_from_keycode**(struct input_dev * dev, unsigned int keycode)

perform sparse keymap lookup

Parameters

struct input_dev * dev Input device using sparse keymap

unsigned int keycode Key code

Description

This function is used to perform `struct key_entry` lookup in an input device using sparse keymap.

int **sparse_keymap_setup**(struct input_dev * dev, const struct key_entry * keymap, int (*setup)(struct input_dev *, struct key_entry *))

set up sparse keymap for an input device

Parameters

struct input_dev * dev Input device

const struct key_entry * keymap Keymap in form of array of `key_entry` structures ending with `KE_END` type entry

int (*)(struct input_dev *, struct key_entry *) setup Function that can be used to adjust keymap entries depending on device's needs, may be `NULL`

Description

The function calculates size and allocates copy of the original keymap after which sets up input device event bits appropriately. The allocated copy of the keymap is automatically freed when it is no longer needed.

void **sparse_keymap_report_entry**(struct input_dev * dev, const struct key_entry * ke, unsigned int value, bool autorelease)
report event corresponding to given key entry

Parameters

struct input_dev * dev Input device for which event should be reported

const struct key_entry * ke key entry describing event

unsigned int value Value that should be reported (ignored by KE_SW entries)

bool autorelease Signals whether release event should be emitted for KE_KEY entries right after reporting press event, ignored by all other entries

Description

This function is used to report input event described by given `struct key_entry`.

bool **sparse_keymap_report_event**(struct input_dev * dev, unsigned int code, unsigned int value, bool autorelease)
report event corresponding to given scancode

Parameters

struct input_dev * dev Input device using sparse keymap

unsigned int code Scan code

unsigned int value Value that should be reported (ignored by KE_SW entries)

bool autorelease Signals whether release event should be emitted for KE_KEY entries right after reporting press event, ignored by all other entries

Description

This function is used to perform lookup in an input device using sparse keymap and report corresponding event. Returns `true` if lookup was successful and `false` otherwise.

CHAPTER

TWENTY

LINUX USB API

20.1 The Linux-USB Host Side API

20.1.1 Introduction to USB on Linux

A Universal Serial Bus (USB) is used to connect a host, such as a PC or workstation, to a number of peripheral devices. USB uses a tree structure, with the host as the root (the system's master), hubs as interior nodes, and peripherals as leaves (and slaves). Modern PCs support several such trees of USB devices, usually a few USB 3.0 (5 GBit/s) or USB 3.1 (10 GBit/s) and some legacy USB 2.0 (480 MBit/s) busses just in case.

That master/slave asymmetry was designed-in for a number of reasons, one being ease of use. It is not physically possible to mistake upstream and downstream or it does not matter with a type C plug (or they are built into the peripheral). Also, the host software doesn't need to deal with distributed auto-configuration since the pre-designated master node manages all that.

Kernel developers added USB support to Linux early in the 2.2 kernel series and have been developing it further since then. Besides support for each new generation of USB, various host controllers gained support, new drivers for peripherals have been added and advanced features for latency measurement and improved power management introduced.

Linux can run inside USB devices as well as on the hosts that control the devices. But USB device drivers running inside those peripherals don't do the same things as the ones running inside hosts, so they've been given a different name: gadget drivers. This document does not cover gadget drivers.

20.1.2 USB Host-Side API Model

Host-side drivers for USB devices talk to the "usbcore" APIs. There are two. One is intended for general-purpose drivers (exposed through driver frameworks), and the other is for drivers that are part of the core. Such core drivers include the hub driver (which manages trees of USB devices) and several different kinds of host controller drivers, which control individual busses.

The device model seen by USB drivers is relatively complex.

- USB supports four kinds of data transfers (control, bulk, interrupt, and isochronous). Two of them (control and bulk) use bandwidth as it's avail-

able, while the other two (interrupt and isochronous) are scheduled to provide guaranteed bandwidth.

- The device description model includes one or more "configurations" per device, only one of which is active at a time. Devices are supposed to be capable of operating at lower than their top speeds and may provide a BOS descriptor showing the lowest speed they remain fully operational at.
- From USB 3.0 on configurations have one or more "functions", which provide a common functionality and are grouped together for purposes of power management.
- Configurations or functions have one or more "interfaces", each of which may have "alternate settings". Interfaces may be standardized by USB "Class" specifications, or may be specific to a vendor or device.

  USB device drivers actually bind to interfaces, not devices. Think of them as "interface drivers", though you may not see many devices where the distinction is important. Most USB devices are simple, with only one function, one configuration, one interface, and one alternate setting.
- Interfaces have one or more "endpoints", each of which supports one type and direction of data transfer such as "bulk out" or "interrupt in". The entire configuration may have up to sixteen endpoints in each direction, allocated as needed among all the interfaces.
- Data transfer on USB is packetized; each endpoint has a maximum packet size. Drivers must often be aware of conventions such as flagging the end of bulk transfers using "short" (including zero length) packets.
- The Linux USB API supports synchronous calls for control and bulk messages. It also supports asynchronous calls for all kinds of data transfer, using request structures called "URBs" (USB Request Blocks).

Accordingly, the USB Core API exposed to device drivers covers quite a lot of territory. You'll probably need to consult the USB 3.0 specification, available online from www.usb.org at no cost, as well as class or device specifications.

The only host-side drivers that actually touch hardware (reading/writing registers, handling IRQs, and so on) are the HCDs. In theory, all HCDs provide the same functionality through the same API. In practice, that's becoming more true, but there are still differences that crop up especially with fault handling on the less common controllers. Different controllers don't necessarily report the same aspects of failures, and recovery from faults (including software-induced ones like unlinking an URB) isn't yet fully consistent. Device driver authors should make a point of doing disconnect testing (while the device is active) with each different host controller driver, to make sure drivers don't have bugs of their own as well as to make sure they aren't relying on some HCD-specific behavior.

20.1.3 USB-Standard Types

In <linux/usb/ch9.h> you will find the USB data types defined in chapter 9 of the USB specification. These data types are used throughout USB, and in APIs including this host side API, gadget APIs, usb character devices and debugfs interfaces.

const char * **usb_ep_type_string**(int ep_type)
Returns human readable-name of the endpoint type.

Parameters

int ep_type The endpoint type to return human-readable name for. If it's not any of the types: USB_ENDPOINT_XFER_{CONTROL, ISOC, BULK, INT}, usually got by usb_endpoint_type(), the string 'unknown' will be returned.

const char * **usb_speed_string**(enum usb_device_speed speed)
Returns human readable-name of the speed.

Parameters

enum usb_device_speed speed The speed to return human-readable name for. If it's not any of the speeds defined in usb_device_speed enum, string for USB_SPEED_UNKNOWN will be returned.

enum usb_device_speed **usb_get_maximum_speed**(struct device * dev)
Get maximum requested speed for a given USB controller.

Parameters

struct device * dev Pointer to the given USB controller device

Description

The function gets the maximum speed string from property "maximum-speed", and returns the corresponding enum usb_device_speed.

const char * **usb_state_string**(enum usb_device_state state)
Returns human readable name for the state.

Parameters

enum usb_device_state state The state to return a human-readable name for. If it's not any of the states devices in usb_device_state_string enum, the string UNKNOWN will be returned.

const char * **usb_decode_ctrl**(char * str, size_t size, __u8 bRequestType, __u8 bRequest, __u16 wValue, __u16 wIndex, __u16 wLength)
Returns human readable representation of control request.

Parameters

char * str buffer to return a human-readable representation of control request. This buffer should have about 200 bytes.

size_t size size of str buffer.

__u8 bRequestType matches the USB bmRequestType field

__u8 bRequest matches the USB bRequest field

__u16 wValue matches the USB wValue field (CPU byte order)

__u16 wIndex matches the USB wIndex field (CPU byte order)

__u16 wLength matches the USB wLength field (CPU byte order)

Description

Function returns decoded, formatted and human-readable description of control request packet.

The usage scenario for this is for tracepoints, so function as a return use the same value as in parameters. This approach allows to use this function in TP_printk

Important: wValue, wIndex, wLength parameters before invoking this function should be processed by le16_to_cpu macro.

20.1.4 Host-Side Data Types and Macros

The host side API exposes several layers to drivers, some of which are more necessary than others. These support lifecycle models for host side drivers and devices, and support passing buffers through usbcore to some HCD that performs the I/O for the device driver.

struct **usb_host_endpoint**
host-side endpoint descriptor and queue

Definition

```
struct usb_host_endpoint {
  struct usb_endpoint_descriptor          desc;
  struct usb_ss_ep_comp_descriptor        ss_ep_comp;
  struct usb_ssp_isoc_ep_comp_descriptor  ssp_isoc_ep_comp;
  struct list_head                urb_list;
  void *hcpriv;
  struct ep_device                *ep_dev;
  unsigned char *extra;
  int extralen;
  int enabled;
  int streams;
};
```

Members

desc descriptor for this endpoint, wMaxPacketSize in native byteorder

ss_ep_comp SuperSpeed companion descriptor for this endpoint

ssp_isoc_ep_comp SuperSpeedPlus isoc companion descriptor for this endpoint

urb_list urbs queued to this endpoint; maintained by usbcore

hcpriv for use by HCD; typically holds hardware dma queue head (QH) with one or more transfer descriptors (TDs) per urb

ep_dev ep_device for sysfs info

extra descriptors following this endpoint in the configuration

extralen how many bytes of “extra” are valid

enabled URBs may be submitted to this endpoint

streams number of USB-3 streams allocated on the endpoint

Description

USB requests are always queued to a given endpoint, identified by a descriptor within an active interface in a given USB configuration.

struct **usb_interface**
what usb device drivers talk to

Definition

```
struct usb_interface {
  struct usb_host_interface *altsetting;
  struct usb_host_interface *cur_altsetting;
  unsigned num_altsetting;
  struct usb_interface_assoc_descriptor *intf_assoc;
  int minor;
  enum usb_interface_condition condition;
  unsigned sysfs_files_created:1;
  unsigned ep_devs_created:1;
  unsigned unregistering:1;
  unsigned needs_remote_wakeup:1;
  unsigned needs_altsetting0:1;
  unsigned needs_binding:1;
  unsigned resetting_device:1;
  unsigned authorized:1;
  struct device dev;
  struct device *usb_dev;
  struct work_struct reset_ws;
};
```

Members

altsetting array of interface structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order.

cur_altsetting the current altsetting.

num_altsetting number of altsettings defined.

intf_assoc interface association descriptor

minor the minor number assigned to this interface, if this interface is bound to a driver that uses the USB major number. If this interface does not use the USB major, this field should be unused. The driver should set this value in the probe() function of the driver, after it has been assigned a minor number from the USB core by calling `usb_register_dev()`.

condition binding state of the interface: not bound, binding (in probe()), bound to a driver, or unbinding (in disconnect())

sysfs_files_created sysfs attributes exist

ep_devs_created endpoint child pseudo-devices exist

unregistering flag set when the interface is being unregistered

needs_remote_wakeup flag set when the driver requires remote-wakeup capability during autosuspend.

needs_altsetting0 flag set when a set-interface request for altsetting 0 has been deferred.

needs_binding flag set when the driver should be re-probed or unbound following a reset or suspend operation it doesn't support.

resetting_device USB core reset the device, so use alt setting 0 as current; needs bandwidth alloc after reset.

authorized This allows to (de)authorize individual interfaces instead a whole device in contrast to the device authorization.

dev driver model's view of this device

usb_dev if an interface is bound to the USB major, this will point to the sysfs representation for that device.

reset_ws Used for scheduling resets from atomic context.

Description

USB device drivers attach to interfaces on a physical device. Each interface encapsulates a single high level function, such as feeding an audio stream to a speaker or reporting a change in a volume control. Many USB devices only have one interface. The protocol used to talk to an interface's endpoints can be defined in a usb "class" specification, or by a product's vendor. The (default) control endpoint is part of every interface, but is never listed among the interface's descriptors.

The driver that is bound to the interface can use standard driver model calls such as dev_get_drvdata() on the dev member of this structure.

Each interface may have alternate settings. The initial configuration of a device sets altsetting 0, but the device driver can change that setting using `usb_set_interface()`. Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-conformant USB devices that use isochronous endpoints will use them in non-default settings.

The USB specification says that alternate setting numbers must run from 0 to one less than the total number of alternate settings. But some devices manage to mess this up, and the structures aren't necessarily stored in numerical order anyhow. Use `usb_altnum_to_altsetting()` to look up an alternate setting in the altsetting array based on its number.

struct **usb_interface_cache**
long-term representation of a device interface

Definition

```
struct usb_interface_cache {
  unsigned num_altsetting;
  struct kref ref;
  struct usb_host_interface altsetting[];
};
```

Members

num_altsetting number of altsettings defined.

ref reference counter.

altsetting variable-length array of interface structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order.

Description

These structures persist for the lifetime of a usb_device, unlike struct usb_interface (which persists only as long as its configuration is installed). The altsetting arrays can be accessed through these structures at any time, permitting comparison of configurations and providing support for the /sys/kernel/debug/usb/devices pseudo-file.

struct **usb_host_config**
representation of a device's configuration

Definition

```
struct usb_host_config {
  struct usb_config_descriptor    desc;
  char *string;
  struct usb_interface_assoc_descriptor *intf_assoc[USB_MAXIADS];
  struct usb_interface *interface[USB_MAXINTERFACES];
  struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];
  unsigned char *extra;
  int extralen;
};
```

Members

desc the device's configuration descriptor.

string pointer to the cached version of the iConfiguration string, if present for this configuration.

intf_assoc list of any interface association descriptors in this config

interface array of pointers to usb_interface structures, one for each interface in the configuration. The number of interfaces is stored in desc.bNumInterfaces. These pointers are valid only while the the configuration is active.

intf_cache array of pointers to usb_interface_cache structures, one for each interface in the configuration. These structures exist for the entire life of the device.

extra pointer to buffer containing all extra descriptors associated with this configuration (those preceding the first interface descriptor).

extralen length of the extra descriptors buffer.

Description

USB devices may have multiple configurations, but only one can be active at any time. Each encapsulates a different operational environment; for example, a dual-speed device would have separate configurations for full-speed and high-speed

operation. The number of configurations available is stored in the device descriptor as bNumConfigurations.

A configuration can contain multiple interfaces. Each corresponds to a different function of the USB device, and all are available whenever the configuration is active. The USB standard says that interfaces are supposed to be numbered from 0 to desc.bNumInterfaces-1, but a lot of devices get this wrong. In addition, the interface array is not guaranteed to be sorted in numerical order. Use `usb_ifnum_to_if()` to look up an interface entry based on its number.

Device drivers should not attempt to activate configurations. The choice of which configuration to install is a policy decision based on such considerations as available power, functionality provided, and the user's desires (expressed through userspace tools). However, drivers can call `usb_reset_configuration()` to reinitialize the current configuration and all its interfaces.

struct **usb_device**
kernel's representation of a USB device

Definition

```
struct usb_device {
  int devnum;
  char devpath[16];
  u32 route;
  enum usb_device_state   state;
  enum usb_device_speed   speed;
  unsigned int            rx_lanes;
  unsigned int            tx_lanes;
  struct usb_tt   *tt;
  int ttport;
  unsigned int toggle[2];
  struct usb_device *parent;
  struct usb_bus *bus;
  struct usb_host_endpoint ep0;
  struct device dev;
  struct usb_device_descriptor descriptor;
  struct usb_host_bos *bos;
  struct usb_host_config *config;
  struct usb_host_config *actconfig;
  struct usb_host_endpoint *ep_in[16];
  struct usb_host_endpoint *ep_out[16];
  char **rawdescriptors;
  unsigned short bus_mA;
  u8 portnum;
  u8 level;
  u8 devaddr;
  unsigned can_submit:1;
  unsigned persist_enabled:1;
  unsigned have_langid:1;
  unsigned authorized:1;
  unsigned authenticated:1;
  unsigned wusb:1;
  unsigned lpm_capable:1;
  unsigned usb2_hw_lpm_capable:1;
  unsigned usb2_hw_lpm_besl_capable:1;
  unsigned usb2_hw_lpm_enabled:1;
```

(continues on next page)

(continued from previous page)

```
  unsigned usb2_hw_lpm_allowed:1;
  unsigned usb3_lpm_u1_enabled:1;
  unsigned usb3_lpm_u2_enabled:1;
  int string_langid;
  char *product;
  char *manufacturer;
  char *serial;
  struct list_head filelist;
  int maxchild;
  u32 quirks;
  atomic_t urbnum;
  unsigned long active_duration;
#ifdef CONFIG_PM;
  unsigned long connect_time;
  unsigned do_remote_wakeup:1;
  unsigned reset_resume:1;
  unsigned port_is_suspended:1;
#endif;
  struct wusb_dev *wusb_dev;
  int slot_id;
  enum usb_device_removable removable;
  struct usb2_lpm_parameters l1_params;
  struct usb3_lpm_parameters u1_params;
  struct usb3_lpm_parameters u2_params;
  unsigned lpm_disable_count;
  u16 hub_delay;
  unsigned use_generic_driver:1;
};
```

Members

devnum device number; address on a USB bus

devpath device ID string for use in messages (e.g., /port/…)

route tree topology hex string for use with xHCI

state device state: configured, not attached, etc.

speed device speed: high/full/low (or error)

rx_lanes number of rx lanes in use, USB 3.2 adds dual-lane support

tx_lanes number of tx lanes in use, USB 3.2 adds dual-lane support

tt Transaction Translator info; used with low/full speed dev, highspeed hub

ttport device port on that tt hub

toggle one bit for each endpoint, with ([0] = IN, [1] = OUT) endpoints

parent our hub, unless we’re the root

bus bus we’re part of

ep0 endpoint 0 data (default control pipe)

dev generic device interface

descriptor USB device descriptor

bos USB device BOS descriptor set

config all of the device's configs

actconfig the active configuration

ep_in array of IN endpoints

ep_out array of OUT endpoints

rawdescriptors raw descriptors for each config

bus_mA Current available from the bus

portnum parent port number (origin 1)

level number of USB hub ancestors

devaddr device address, XHCI: assigned by HW, others: same as devnum

can_submit URBs may be submitted

persist_enabled USB_PERSIST enabled for this device

have_langid whether string_langid is valid

authorized policy has said we can use it; (user space) policy determines if we authorize this device to be used or not. By default, wired USB devices are authorized. WUSB devices are not, until we authorize them from user space. FIXME – complete doc

authenticated Crypto authentication passed

wusb device is Wireless USB

lpm_capable device supports LPM

usb2_hw_lpm_capable device can perform USB2 hardware LPM

usb2_hw_lpm_besl_capable device can perform USB2 hardware BESL LPM

usb2_hw_lpm_enabled USB2 hardware LPM is enabled

usb2_hw_lpm_allowed Userspace allows USB 2.0 LPM to be enabled

usb3_lpm_u1_enabled USB3 hardware U1 LPM enabled

usb3_lpm_u2_enabled USB3 hardware U2 LPM enabled

string_langid language ID for strings

product iProduct string, if present (static)

manufacturer iManufacturer string, if present (static)

serial iSerialNumber string, if present (static)

filelist usbfs files that are open to this device

maxchild number of ports if hub

quirks quirks of the whole device

urbnum number of URBs submitted for the whole device

active_duration total time device is not suspended

connect_time time device was first connected

do_remote_wakeup remote wakeup should be enabled

reset_resume needs reset instead of resume

port_is_suspended the upstream port is suspended (L2 or U3)

wusb_dev if this is a Wireless USB device, link to the WUSB specific data for the device.

slot_id Slot ID assigned by xHCI

removable Device can be physically removed from this port

l1_params best effor service latency for USB2 L1 LPM state, and L1 timeout.

u1_params exit latencies for USB3 U1 LPM state, and hub-initiated timeout.

u2_params exit latencies for USB3 U2 LPM state, and hub-initiated timeout.

lpm_disable_count Ref count used by usb_disable_lpm() and usb_enable_lpm() to keep track of the number of functions that require USB 3.0 Link Power Management to be disabled for this usb_device. This count should only be manipulated by those functions, with the bandwidth_mutex is held.

hub_delay cached value consisting of: parent->hub_delay + wHubDelay + tTPTransmissionDelay (40ns)

Will be used as wValue for SetIsochDelay requests.

Notes

Usbcore drivers should not set usbdev->state directly. Instead use `usb_set_device_state()`.

usb_hub_for_each_child(hdev, port1, child)
iterate over all child devices on the hub

Parameters

hdev USB device belonging to the usb hub

port1 portnum associated with child device

child child device pointer

int **usb_interface_claimed**(struct usb_interface * iface)
returns true iff an interface is claimed

Parameters

struct usb_interface * iface the interface being checked

Return

`true` (nonzero) iff the interface is claimed, else `false` (zero).

Note

Callers must own the driver model's usb bus readlock. So driver probe() entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

int **usb_make_path**(struct usb_device * dev, char * buf, size_t size)
returns stable device path in the usb tree

Parameters

struct usb_device * dev the device whose path is being constructed

char * buf where to put the string

size_t size how big is "buf"?

Return

Length of the string (> 0) or negative if size was too small.

Note

This identifier is intended to be "stable", reflecting physical paths in hardware such as physical bus addresses for host controllers or ports on USB hubs. That makes it stay the same until systems are physically reconfigured, by re-cabling a tree of USB devices or by moving USB host controllers. Adding and removing devices, including virtual root hubs in host controller driver modules, does not change these path identifiers; neither does rebooting or re-enumerating. These are more useful identifiers than changeable ("unstable") ones like bus numbers or device addresses.

Description

With a partial exception for devices connected to USB 2.0 root hubs, these identifiers are also predictable. So long as the device tree isn't changed, plugging any USB device into a given hub port always gives it the same path. Because of the use of "companion" controllers, devices connected to ports on USB 2.0 root hubs (EHCI host controllers) will get one path ID if they are high speed, and a different one if they are full or low speed.

USB_DEVICE(vend, prod)
macro used to describe a specific usb device

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

Description

This macro is used to create a struct usb_device_id that matches a specific device.

USB_DEVICE_VER(vend, prod, lo, hi)
describe a specific usb device with a version range

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

lo the bcdDevice_lo value

hi the bcdDevice_hi value

Description

This macro is used to create a struct usb_device_id that matches a specific device, with a version range.

USB_DEVICE_INTERFACE_CLASS(vend, prod, cl)
describe a usb device with a specific interface class

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

cl bInterfaceClass value

Description

This macro is used to create a struct usb_device_id that matches a specific interface class of devices.

USB_DEVICE_INTERFACE_PROTOCOL(vend, prod, pr)
describe a usb device with a specific interface protocol

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

pr bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific interface protocol of devices.

USB_DEVICE_INTERFACE_NUMBER(vend, prod, num)
describe a usb device with a specific interface number

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

num bInterfaceNumber value

Description

This macro is used to create a struct usb_device_id that matches a specific interface number of devices.

USB_DEVICE_INFO(cl, sc, pr)
macro used to describe a class of usb devices

Parameters

cl bDeviceClass value

sc bDeviceSubClass value

pr bDeviceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific class of devices.

USB_INTERFACE_INFO(cl, sc, pr)
macro used to describe a class of usb interfaces

Parameters

cl bInterfaceClass value

sc bInterfaceSubClass value

pr bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific class of interfaces.

USB_DEVICE_AND_INTERFACE_INFO(vend, prod, cl, sc, pr)
describe a specific usb device with a class of usb interfaces

Parameters

vend the 16 bit USB Vendor ID

prod the 16 bit USB Product ID

cl bInterfaceClass value

sc bInterfaceSubClass value

pr bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific device with a specific class of interfaces.

This is especially useful when explicitly matching devices that have vendor specific bDeviceClass values, but standards-compliant interfaces.

USB_VENDOR_AND_INTERFACE_INFO(vend, cl, sc, pr)
describe a specific usb vendor with a class of usb interfaces

Parameters

vend the 16 bit USB Vendor ID

cl bInterfaceClass value

sc bInterfaceSubClass value

pr bInterfaceProtocol value

Description

This macro is used to create a struct usb_device_id that matches a specific vendor with a specific class of interfaces.

This is especially useful when explicitly matching devices that have vendor specific bDeviceClass values, but standards-compliant interfaces.

struct **usbdrv_wrap**
wrapper for driver-model structure

Definition

```
struct usbdrv_wrap {
  struct device_driver driver;
  int for_devices;
};
```

Members

driver The driver-model core driver structure.

for_devices Non-zero for device drivers, 0 for interface drivers.

struct **usb_driver**
identifies USB interface driver to usbcore

Definition

```
struct usb_driver {
  const char *name;
  int (*probe) (struct usb_interface *intf, const struct usb_device_id␣
↪*id);
  void (*disconnect) (struct usb_interface *intf);
  int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,␣
↪void *buf);
  int (*suspend) (struct usb_interface *intf, pm_message_t message);
  int (*resume) (struct usb_interface *intf);
  int (*reset_resume)(struct usb_interface *intf);
  int (*pre_reset)(struct usb_interface *intf);
  int (*post_reset)(struct usb_interface *intf);
  const struct usb_device_id *id_table;
  const struct attribute_group **dev_groups;
  struct usb_dynids dynids;
  struct usbdrv_wrap drvwrap;
  unsigned int no_dynamic_id:1;
  unsigned int supports_autosuspend:1;
  unsigned int disable_hub_initiated_lpm:1;
  unsigned int soft_unbind:1;
};
```

Members

name The driver name should be unique among USB drivers, and should normally be the same as the module name.

probe Called to see if the driver is willing to manage a particular interface on a device. If it is, probe returns zero and uses usb_set_intfdata() to associate driver-specific data with the interface. It may also use `usb_set_interface()` to specify the appropriate altsetting. If unwilling to manage the interface, return -ENODEV, if genuine IO errors occurred, an appropriate negative errno value.

disconnect Called when the interface is no longer accessible, usually because its device has been (or is being) disconnected or the driver module is being unloaded.

unlocked_ioctl Used for drivers that want to talk to userspace through the "usbfs" filesystem. This lets devices provide ways to expose information to user space regardless of where they do (or don't) show up otherwise in the filesystem.

suspend Called when the device is going to be suspended by the system either from system sleep or runtime suspend context. The return value will be ignored in system sleep context, so do NOT try to continue using the device if suspend fails in this case. Instead, let the resume or reset-resume routine recover from the failure.

resume Called when the device is being resumed by the system.

reset_resume Called when the suspended device has been reset instead of being resumed.

pre_reset Called by `usb_reset_device()` when the device is about to be reset. This routine must not return until the driver has no active URBs for the device, and no more URBs may be submitted until the post_reset method is called.

post_reset Called by `usb_reset_device()` after the device has been reset

id_table USB drivers use ID table to support hotplugging. Export this with MODULE_DEVICE_TABLE(usb,⋯). This must be set or your driver's probe function will never get called.

dev_groups Attributes attached to the device that will be created once it is bound to the driver.

dynids used internally to hold the list of dynamically added device ids for this driver.

drvwrap Driver-model core structure wrapper.

no_dynamic_id if set to 1, the USB core will not allow dynamic ids to be added to this driver by preventing the sysfs file from being created.

supports_autosuspend if set to 0, the USB core will not allow autosuspend for interfaces bound to this driver.

disable_hub_initiated_lpm if set to 1, the USB core will not allow hubs to initiate lower power link state transitions when an idle timeout occurs. Device-initiated USB 3.0 link PM will still be allowed.

soft_unbind if set to 1, the USB core will not kill URBs and disable endpoints before calling the driver's disconnect method.

Description

USB interface drivers must provide a name, probe() and disconnect() methods, and an id_table. Other driver fields are optional.

The id_table is used in hotplugging. It holds a set of descriptors, and specialized data may be associated with each entry. That table is used by both user and kernel mode hotplugging support.

The probe() and disconnect() methods are called in a context where they can sleep, but they should avoid abusing the privilege. Most work to connect to a device should be done when the device is opened, and undone at the last close. The disconnect code needs to address concurrency issues with respect to open() and

close() methods, as well as forcing all pending I/O requests to complete (by unlinking them as necessary, and blocking until the unlinks complete).

struct **usb_device_driver**
identifies USB device driver to usbcore

Definition

```
struct usb_device_driver {
  const char *name;
  bool (*match) (struct usb_device *udev);
  int (*probe) (struct usb_device *udev);
  void (*disconnect) (struct usb_device *udev);
  int (*suspend) (struct usb_device *udev, pm_message_t message);
  int (*resume) (struct usb_device *udev, pm_message_t message);
  const struct attribute_group **dev_groups;
  struct usbdrv_wrap drvwrap;
  const struct usb_device_id *id_table;
  unsigned int supports_autosuspend:1;
  unsigned int generic_subclass:1;
};
```

Members

name The driver name should be unique among USB drivers, and should normally be the same as the module name.

probe Called to see if the driver is willing to manage a particular device. If it is, probe returns zero and uses dev_set_drvdata() to associate driver-specific data with the device. If unwilling to manage the device, return a negative errno value.

disconnect Called when the device is no longer accessible, usually because it has been (or is being) disconnected or the driver's module is being unloaded.

suspend Called when the device is going to be suspended by the system.

resume Called when the device is being resumed by the system.

dev_groups Attributes attached to the device that will be created once it is bound to the driver.

drvwrap Driver-model core structure wrapper.

supports_autosuspend if set to 0, the USB core will not allow autosuspend for devices bound to this driver.

generic_subclass if set to 1, the generic USB driver's probe, disconnect, resume and suspend functions will be called in addition to the driver's own, so this part of the setup does not need to be replicated.

Description

USB drivers must provide all the fields listed above except drvwrap.

struct **usb_class_driver**
identifies a USB driver that wants to use the USB major number

Definition

```
struct usb_class_driver {
  char *name;
  char *(*devnode)(struct device *dev, umode_t *mode);
  const struct file_operations *fops;
  int minor_base;
};
```

Members

name the usb class device name for this driver. Will show up in sysfs.

devnode Callback to provide a naming hint for a possible device node to create.

fops pointer to the struct file_operations of this driver.

minor_base the start of the minor range for this driver.

Description

This structure is used for the `usb_register_dev()` and `usb_deregister_dev()` functions, to consolidate a number of the parameters used for them.

module_usb_driver(__usb_driver)
Helper macro for registering a USB driver

Parameters

__usb_driver usb_driver struct

Description

Helper macro for USB drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init()` and `module_exit()`

struct **urb**
USB Request Block

Definition

```
struct urb {
  struct list_head urb_list;
  struct list_head anchor_list;
  struct usb_anchor *anchor;
  struct usb_device *dev;
  struct usb_host_endpoint *ep;
  unsigned int pipe;
  unsigned int stream_id;
  int status;
  unsigned int transfer_flags;
  void *transfer_buffer;
  dma_addr_t transfer_dma;
  struct scatterlist *sg;
  int num_mapped_sgs;
  int num_sgs;
  u32 transfer_buffer_length;
  u32 actual_length;
  unsigned char *setup_packet;
  dma_addr_t setup_dma;
  int start_frame;
```

(continues on next page)

(continued from previous page)

```
  int number_of_packets;
  int interval;
  int error_count;
  void *context;
  usb_complete_t complete;
  struct usb_iso_packet_descriptor iso_frame_desc[];
};
```

Members

urb_list For use by current owner of the URB.

anchor_list membership in the list of an anchor

anchor to anchor URBs to a common mooring

dev Identifies the USB device to perform the request.

ep Points to the endpoint's data structure. Will eventually replace **pipe**.

pipe Holds endpoint number, direction, type, and more. Create these values with the eight macros available; usb_{snd,rcv}TYPEpipe(dev,endpoint), where the TYPE is "ctrl" (control), "bulk", "int" (interrupt), or "iso" (isochronous). For example usb_sndbulkpipe() or usb_rcvintpipe(). Endpoint numbers range from zero to fifteen. Note that "in" endpoint two is a different endpoint (and pipe) from "out" endpoint two. The current configuration controls the existence, type, and maximum packet size of any given endpoint.

stream_id the endpoint's stream ID for bulk streams

status This is read in non-iso completion functions to get the status of the particular request. ISO requests only use it to tell whether the URB was unlinked; detailed status for each frame is in the fields of the iso_frame-desc.

transfer_flags A variety of flags may be used to affect how URB submission, unlinking, or operation are handled. Different kinds of URB can use different flags.

transfer_buffer This identifies the buffer to (or from) which the I/O request will be performed unless URB_NO_TRANSFER_DMA_MAP is set (however, do not leave garbage in transfer_buffer even then). This buffer must be suitable for DMA; allocate it with kmalloc() or equivalent. For transfers to "in" endpoints, contents of this buffer will be modified. This buffer is used for the data stage of control transfers.

transfer_dma When transfer_flags includes URB_NO_TRANSFER_DMA_MAP, the device driver is saying that it provided this DMA address, which the host controller driver should use in preference to the transfer_buffer.

sg scatter gather buffer list, the buffer size of each element in the list (except the last) must be divisible by the endpoint's max packet size if no_sg_constraint isn't set in 'struct usb_bus'

num_mapped_sgs (internal) number of mapped sg entries

num_sgs number of entries in the sg list

transfer_buffer_length How big is transfer_buffer. The transfer may be broken up into chunks according to the current maximum packet size for the endpoint, which is a function of the configuration and is encoded in the pipe. When the length is zero, neither transfer_buffer nor transfer_dma is used.

actual_length This is read in non-iso completion functions, and it tells how many bytes (out of transfer_buffer_length) were transferred. It will normally be the same as requested, unless either an error was reported or a short read was performed. The URB_SHORT_NOT_OK transfer flag may be used to make such short reads be reported as errors.

setup_packet Only used for control transfers, this points to eight bytes of setup data. Control transfers always start by sending this data to the device. Then transfer_buffer is read or written, if needed.

setup_dma DMA pointer for the setup packet. The caller must not use this field; setup_packet must point to a valid buffer.

start_frame Returns the initial frame for isochronous transfers.

number_of_packets Lists the number of ISO transfer buffers.

interval Specifies the polling interval for interrupt or isochronous transfers. The units are frames (milliseconds) for full and low speed devices, and microframes (1/8 millisecond) for highspeed and SuperSpeed devices.

error_count Returns the number of ISO transfers that reported errors.

context For use in completion functions. This normally points to request-specific driver context.

complete Completion handler. This URB is passed as the parameter to the completion function. The completion function may then do what it likes with the URB, including resubmitting or freeing it.

iso_frame_desc Used to provide arrays of ISO transfer buffers and to collect the transfer status for each buffer.

Description

This structure identifies USB transfer requests. URBs must be allocated by calling `usb_alloc_urb()` and freed with a call to `usb_free_urb()`. Initialization may be done using various usb_fill_*_urb() functions. URBs are submitted using `usb_submit_urb()`, and pending requests may be canceled using `usb_unlink_urb()` or `usb_kill_urb()`.

Data Transfer Buffers:

Normally drivers provide I/O buffers allocated with kmalloc() or otherwise taken from the general page pool. That is provided by transfer_buffer (control requests also use setup_packet), and host controller drivers perform a dma mapping (and unmapping) for each buffer transferred. Those mapping operations can be expensive on some platforms (perhaps using a dma bounce buffer or talking to an IOMMU), although they' re cheap on commodity x86 and ppc hardware.

Alternatively, drivers may pass the URB_NO_TRANSFER_DMA_MAP transfer flag, which tells the host controller driver that no such mapping is needed for the transfer_buffer since the device driver is DMA-aware. For example, a device driver might allocate a DMA buffer with `usb_alloc_coherent()` or call usb_buffer_map().

When this transfer flag is provided, host controller drivers will attempt to use the dma address found in the transfer_dma field rather than determining a dma address themselves.

Note that transfer_buffer must still be set if the controller does not support DMA (as indicated by hcd_uses_dma()) and when talking to root hub. If you have to trasfer between highmem zone and the device on such controller, create a bounce buffer or bail out with an error. If transfer_buffer cannot be set (is in highmem) and the controller is DMA capable, assign NULL to it, so that usbmon knows not to use the value. The setup_packet must always be set, so it cannot be located in highmem.

Initialization:

All URBs submitted must initialize the dev, pipe, transfer_flags (may be zero), and complete fields. All URBs must also initialize transfer_buffer and transfer_buffer_length. They may provide the URB_SHORT_NOT_OK transfer flag, indicating that short reads are to be treated as errors; that flag is invalid for write requests.

Bulk URBs may use the URB_ZERO_PACKET transfer flag, indicating that bulk OUT transfers should always terminate with a short packet, even if it means adding an extra zero length packet.

Control URBs must provide a valid pointer in the setup_packet field. Unlike the transfer_buffer, the setup_packet may not be mapped for DMA beforehand.

Interrupt URBs must provide an interval, saying how often (in milliseconds or, for highspeed devices, 125 microsecond units) to poll for transfers. After the URB has been submitted, the interval field reflects how the transfer was actually scheduled. The polling interval may be more frequent than requested. For example, some controllers have a maximum interval of 32 milliseconds, while others support intervals of up to 1024 milliseconds. Isochronous URBs also have transfer intervals. (Note that for isochronous endpoints, as well as high speed interrupt endpoints, the encoding of the transfer interval in the endpoint descriptor is logarithmic. Device drivers must convert that value to linear units themselves.)

If an isochronous endpoint queue isn't already running, the host controller will schedule a new URB to start as soon as bandwidth utilization allows. If the queue is running then a new URB will be scheduled to start in the first transfer slot following the end of the preceding URB, if that slot has not already expired. If the slot has expired (which can happen when IRQ delivery is delayed for a long time), the scheduling behavior depends on the URB_ISO_ASAP flag. If the flag is clear then the URB will be scheduled to start in the expired slot, implying that some of its packets will not be transferred; if the flag is set then the URB will be scheduled in the first unexpired slot, breaking the queue's synchronization. Upon URB completion, the start_frame field will be set to the (micro)frame number in which the transfer was scheduled. Ranges for frame counter values are HC-specific and can go from as low as 256 to as high as 65536 frames.

Isochronous URBs have a different data transfer model, in part because the quality of service is only "best effort". Callers provide specially allocated URBs, with number_of_packets worth of iso_frame_desc structures at the end. Each such packet is an individual ISO transfer. Isochronous URBs are normally queued, submitted by drivers to arrange that transfers are at least double buffered, and then explicitly

resubmitted in completion handlers, so that data (such as audio or video) streams at as constant a rate as the host controller scheduler can support.

Completion Callbacks:

The completion callback is made in_interrupt(), and one of the first things that a completion handler should do is check the status field. The status field is provided for all URBs. It is used to report unlinked URBs, and status for all non-ISO transfers. It should not be examined before the URB is returned to the completion handler.

The context field is normally used to link URBs back to the relevant driver or request state.

When the completion callback is invoked for non-isochronous URBs, the actual_length field tells how many bytes were transferred. This field is updated even when the URB terminated with an error or was unlinked.

ISO transfer status is reported in the status and actual_length fields of the iso_frame_desc array, and the number of errors is reported in error_count. Completion callbacks for ISO transfers will normally (re)submit URBs to ensure a constant transfer rate.

Note that even fields marked "public" should not be touched by the driver when the urb is owned by the hcd, that is, since the call to `usb_submit_urb()` till the entry into the completion routine.

void **usb_fill_control_urb**(struct urb * urb, struct usb_device * dev, unsigned int pipe, unsigned char * setup_packet, void * transfer_buffer, int buffer_length, usb_complete_t complete_fn, void * context)

initializes a control urb

Parameters

struct urb * urb pointer to the urb to initialize.

struct usb_device * dev pointer to the struct usb_device for this urb.

unsigned int pipe the endpoint pipe

unsigned char * setup_packet pointer to the setup_packet buffer

void * transfer_buffer pointer to the transfer buffer

int buffer_length length of the transfer buffer

usb_complete_t complete_fn pointer to the usb_complete_t function

void * context what to set the urb context to.

Description

Initializes a control urb with the proper information needed to submit it to a device.

void **usb_fill_bulk_urb**(struct urb * urb, struct usb_device * dev, unsigned int pipe, void * transfer_buffer, int buffer_length, usb_complete_t complete_fn, void * context)

macro to help initialize a bulk urb

Parameters

struct urb * urb pointer to the urb to initialize.

struct usb_device * dev pointer to the struct usb_device for this urb.

unsigned int pipe the endpoint pipe

void * transfer_buffer pointer to the transfer buffer

int buffer_length length of the transfer buffer

usb_complete_t complete_fn pointer to the usb_complete_t function

void * context what to set the urb context to.

Description

Initializes a bulk urb with the proper information needed to submit it to a device.

void **usb_fill_int_urb**(struct urb * urb, struct usb_device * dev, unsigned int pipe, void * transfer_buffer, int buffer_length, usb_complete_t complete_fn, void * context, int interval)
macro to help initialize a interrupt urb

Parameters

struct urb * urb pointer to the urb to initialize.

struct usb_device * dev pointer to the struct usb_device for this urb.

unsigned int pipe the endpoint pipe

void * transfer_buffer pointer to the transfer buffer

int buffer_length length of the transfer buffer

usb_complete_t complete_fn pointer to the usb_complete_t function

void * context what to set the urb context to.

int interval what to set the urb interval to, encoded like the endpoint descriptor's bInterval value.

Description

Initializes a interrupt urb with the proper information needed to submit it to a device.

Note that High Speed and SuperSpeed(+) interrupt endpoints use a logarithmic encoding of the endpoint interval, and express polling intervals in microframes (eight per millisecond) rather than in frames (one per millisecond).

Wireless USB also uses the logarithmic encoding, but specifies it in units of 128us instead of 125us. For Wireless USB devices, the interval is passed through to the host controller, rather than being translated into microframe units.

int **usb_urb_dir_in**(struct urb * urb)
check if an URB describes an IN transfer

Parameters

struct urb * urb URB to be checked

Return

1 if **urb** describes an IN transfer (device-to-host), otherwise 0.

int **usb_urb_dir_out**(struct urb * urb)
check if an URB describes an OUT transfer

Parameters

struct urb * urb URB to be checked

Return

1 if **urb** describes an OUT transfer (host-to-device), otherwise 0.

struct **usb_sg_request**
support for scatter/gather I/O

Definition

```
struct usb_sg_request {
  int status;
  size_t bytes;
};
```

Members

status zero indicates success, else negative errno

bytes counts bytes transferred.

Description

These requests are initialized using `usb_sg_init()`, and then are used as request handles passed to `usb_sg_wait()` or `usb_sg_cancel()`. Most members of the request object aren't for driver access.

The status and bytecount values are valid only after `usb_sg_wait()` returns. If the status is zero, then the bytecount matches the total from the request.

After an error completion, drivers may need to clear a halt condition on the endpoint.

20.1.5 USB Core APIs

There are two basic I/O models in the USB API. The most elemental one is asynchronous: drivers submit requests in the form of an URB, and the URB's completion callback handles the next step. All USB transfer types support that model, although there are special cases for control URBs (which always have setup and status stages, but may not have a data stage) and isochronous URBs (which allow large packets and include per-packet fault reports). Built on top of that is synchronous API support, where a driver calls a routine that allocates one or more URBs, submits them, and waits until they complete. There are synchronous wrappers for single-buffer control and bulk transfers (which are awkward to use in some driver disconnect scenarios), and for scatterlist based streaming i/o (bulk or interrupt).

USB drivers need to provide buffers that can be used for DMA, although they don't necessarily need to provide the DMA mapping themselves. There are APIs to

use used when allocating DMA buffers, which can prevent use of bounce buffers on some systems. In some cases, drivers may be able to rely on 64bit DMA to eliminate another kind of bounce buffer.

void **usb_init_urb**(struct urb * urb)
initializes a urb so that it can be used by a USB driver

Parameters

struct urb * urb pointer to the urb to initialize

Description

Initializes a urb so that the USB subsystem can use it properly.

If a urb is created with a call to `usb_alloc_urb()` it is not necessary to call this function. Only use this if you allocate the space for a struct urb on your own. If you call this function, be careful when freeing the memory for your urb that it is no longer in use by the USB core.

Only use this function if you _really_ understand what you are doing.

struct urb * **usb_alloc_urb**(int iso_packets, gfp_t mem_flags)
creates a new urb for a USB driver to use

Parameters

int iso_packets number of iso packets for this urb

gfp_t mem_flags the type of memory to allocate, see kmalloc() for a list of valid options for this.

Description

Creates an urb for the USB driver to use, initializes a few internal structures, increments the usage counter, and returns a pointer to it.

If the driver want to use this urb for interrupt, control, or bulk endpoints, pass ‘0’ as the number of iso packets.

The driver must call `usb_free_urb()` when it is finished with the urb.

Return

A pointer to the new urb, or `NULL` if no memory is available.

void **usb_free_urb**(struct urb * urb)
frees the memory used by a urb when all users of it are finished

Parameters

struct urb * urb pointer to the urb to free, may be NULL

Description

Must be called when a user of a urb is finished with it. When the last user of the urb calls this function, the memory of the urb is freed.

Note

The transfer buffer associated with the urb is not freed unless the URB_FREE_BUFFER transfer flag is set.

struct urb * **usb_get_urb**(struct urb * urb)
increments the reference count of the urb

Parameters

struct urb * urb pointer to the urb to modify, may be NULL

Description

This must be called whenever a urb is transferred from a device driver to a host controller driver. This allows proper reference counting to happen for urbs.

Return

A pointer to the urb with the incremented reference counter.

void **usb_anchor_urb**(struct urb * urb, struct usb_anchor * anchor)
anchors an URB while it is processed

Parameters

struct urb * urb pointer to the urb to anchor

struct usb_anchor * anchor pointer to the anchor

Description

This can be called to have access to URBs which are to be executed without bothering to track them

void **usb_unanchor_urb**(struct urb * urb)
unanchors an URB

Parameters

struct urb * urb pointer to the urb to anchor

Description

Call this to stop the system keeping track of this URB

int **usb_urb_ep_type_check**(const struct urb * urb)
sanity check of endpoint in the given urb

Parameters

const struct urb * urb urb to be checked

Description

This performs a light-weight sanity check for the endpoint in the given urb. It returns 0 if the urb contains a valid endpoint, otherwise a negative error code.

int **usb_submit_urb**(struct urb * urb, gfp_t mem_flags)
issue an asynchronous transfer request for an endpoint

Parameters

struct urb * urb pointer to the urb describing the request

gfp_t mem_flags the type of memory to allocate, see kmalloc() for a list of valid options for this.

Description

This submits a transfer request, and transfers control of the URB describing that request to the USB subsystem. Request completion will be indicated later, asynchronously, by calling the completion handler. The three types of completion are success, error, and unlink (a software-induced fault, also called "request cancellation").

URBs may be submitted in interrupt context.

The caller must have correctly initialized the URB before submitting it. Functions such as `usb_fill_bulk_urb()` and `usb_fill_control_urb()` are available to ensure that most fields are correctly initialized, for the particular kind of transfer, although they will not initialize any transfer flags.

If the submission is successful, the complete() callback from the URB will be called exactly once, when the USB core and Host Controller Driver (HCD) are finished with the URB. When the completion function is called, control of the URB is returned to the device driver which issued the request. The completion handler may then immediately free or reuse that URB.

With few exceptions, USB device drivers should never access URB fields provided by usbcore or the HCD until its complete() is called. The exceptions relate to periodic transfer scheduling. For both interrupt and isochronous urbs, as part of successful URB submission urb->interval is modified to reflect the actual transfer period used (normally some power of two units). And for isochronous urbs, urb->start_frame is modified to reflect when the URB's transfers were scheduled to start.

Not all isochronous transfer scheduling policies will work, but most host controller drivers should easily handle ISO queues going from now until 10-200 msec into the future. Drivers should try to keep at least one or two msec of data in the queue; many controllers require that new transfers start at least 1 msec in the future when they are added. If the driver is unable to keep up and the queue empties out, the behavior for new submissions is governed by the URB_ISO_ASAP flag. If the flag is set, or if the queue is idle, then the URB is always assigned to the first available (and not yet expired) slot in the endpoint's schedule. If the flag is not set and the queue is active then the URB is always assigned to the next slot in the schedule following the end of the endpoint's previous URB, even if that slot is in the past. When a packet is assigned in this way to a slot that has already expired, the packet is not transmitted and the corresponding usb_iso_packet_descriptor's status field will return -EXDEV. If this would happen to all the packets in the URB, submission fails with a -EXDEV error code.

For control endpoints, the synchronous `usb_control_msg()` call is often used (in non-interrupt context) instead of this call. That is often used through convenience wrappers, for the requests that are standardized in the USB 2.0 specification. For bulk endpoints, a synchronous `usb_bulk_msg()` call is available.

Request Queuing:

URBs may be submitted to endpoints before previous ones complete, to minimize the impact of interrupt latencies and system overhead on data throughput. With that queuing policy, an endpoint's queue would never be empty. This is required for continuous isochronous data streams, and may also be required for some kinds of interrupt transfers. Such queuing also maximizes bandwidth utilization by letting

USB controllers start work on later requests before driver software has finished the completion processing for earlier (successful) requests.

As of Linux 2.6, all USB endpoint transfer queues support depths greater than one. This was previously a HCD-specific behavior, except for ISO transfers. Non-isochronous endpoint queues are inactive during cleanup after faults (transfer errors or cancellation).

Reserved Bandwidth Transfers:

Periodic transfers (interrupt or isochronous) are performed repeatedly, using the interval specified in the urb. Submitting the first urb to the endpoint reserves the bandwidth necessary to make those transfers. If the USB subsystem can't allocate sufficient bandwidth to perform the periodic request, submitting such a periodic request should fail.

For devices under xHCI, the bandwidth is reserved at configuration time, or when the alt setting is selected. If there is not enough bus bandwidth, the configuration/alt setting request will fail. Therefore, submissions to periodic endpoints on devices under xHCI should never fail due to bandwidth constraints.

Device drivers must explicitly request that repetition, by ensuring that some URB is always on the endpoint's queue (except possibly for short periods during completion callbacks). When there is no longer an urb queued, the endpoint's bandwidth reservation is canceled. This means drivers can use their completion handlers to ensure they keep bandwidth they need, by reinitializing and resubmitting the just-completed urb until the driver longer needs that periodic bandwidth.

Memory Flags:

The general rules for how to decide which mem_flags to use are the same as for kmalloc. There are four different possible values; GFP_KERNEL, GFP_NOFS, GFP_NOIO and GFP_ATOMIC.

GFP_NOFS is not ever used, as it has not been implemented yet.

GFP_ATOMIC is used when

(a) you are inside a completion handler, an interrupt, bottom half, tasklet or timer, or

(b) you are holding a spinlock or rwlock (does not apply to semaphores), or

(c) current->state != TASK_RUNNING, this is the case only after you've changed it.

GFP_NOIO is used in the block io path and error handling of storage devices.

All other situations use GFP_KERNEL.

Some more specific rules for mem_flags can be inferred, such as

(1) start_xmit, timeout, and receive methods of network drivers must use GFP_ATOMIC (they are called with a spinlock held);

(2) queuecommand methods of scsi drivers must use GFP_ATOMIC (also called with a spinlock held);

(3) If you use a kernel thread with a network driver you must use GFP_NOIO, unless (b) or (c) apply;

(4) after you have done a down() you can use GFP_KERNEL, unless (b) or (c) apply or your are in a storage driver's block io path;

(5) USB probe and disconnect can use GFP_KERNEL unless (b) or (c) apply; and

(6) changing firmware on a running storage or net device uses GFP_NOIO, unless b) or c) apply

Return

0 on successful submissions. A negative error number otherwise.

int **usb_unlink_urb**(struct urb * urb)
abort/cancel a transfer request for an endpoint

Parameters

struct urb * urb pointer to urb describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. URBs complete only once per submission, and may be canceled only once per submission. Successful cancellation means termination of **urb** will be expedited and the completion handler will be called with a status code indicating that the request has been canceled (rather than any other code).

Drivers should not call this routine or related routines, such as `usb_kill_urb()` or `usb_unlink_anchored_urbs()`, after their disconnect method has returned. The disconnect function should synchronize with a driver's I/O routines to insure that all URB-related activity has completed before it returns.

This request is asynchronous, however the HCD might call the ->complete() callback during unlink. Therefore when drivers call `usb_unlink_urb()`, they must not hold any locks that may be taken by the completion function. Success is indicated by returning -EINPROGRESS, at which time the URB will probably not yet have been given back to the device driver. When it is eventually called, the completion function will see **urb->status** == -ECONNRESET. Failure is indicated by `usb_unlink_urb()` returning any other value. Unlinking will fail when **urb** is not currently "linked" (i.e., it was never submitted, or it was unlinked before, or the hardware is already finished with it), even if the completion handler has not yet run.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

Unlinking and Endpoint Queues:

[The behaviors and guarantees described below do not apply to virtual root hubs but only to endpoint queues for physical USB devices.]

Host Controller Drivers (HCDs) place all the URBs for a particular endpoint in a queue. Normally the queue advances as the controller hardware processes each request. But when an URB terminates with an error its queue generally stops (see below), at least until that URB's completion routine returns. It is guaranteed that a stopped queue will not restart until all its unlinked URBs have been fully

retired, with their completion routines run, even if that's not until some time after the original completion handler returns. The same behavior and guarantee apply when an URB terminates because it was unlinked.

Bulk and interrupt endpoint queues are guaranteed to stop whenever an URB terminates with any sort of error, including -ECONNRESET, -ENOENT, and -EREMOTEIO. Control endpoint queues behave the same way except that they are not guaranteed to stop for -EREMOTEIO errors. Queues for isochronous endpoints are treated differently, because they must advance at fixed rates. Such queues do not stop when an URB encounters an error or is unlinked. An unlinked isochronous URB may leave a gap in the stream of packets; it is undefined whether such gaps can be filled in.

Note that early termination of an URB because a short packet was received will generate a -EREMOTEIO error if and only if the URB_SHORT_NOT_OK flag is set. By setting this flag, USB device drivers can build deep queues for large or complex bulk transfers and clean them up reliably after any sort of aborted transfer by unlinking all pending URBs at the first fault.

When a control URB terminates with an error other than -EREMOTEIO, it is quite likely that the status stage of the transfer will not take place.

Return

-EINPROGRESS on success. See description for other values on failure.

void **usb_kill_urb**(struct urb * urb)
cancel a transfer request and wait for it to finish

Parameters

struct urb * urb pointer to URB describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. It is guaranteed that upon return all completion handlers will have finished and the URB will be totally idle and available for reuse. These features make this an ideal way to stop I/O in a disconnect() callback or close() function. If the request has not already finished or been unlinked the completion handler will see urb->status == -ENOENT.

While the routine is running, attempts to resubmit the URB will fail with error -EPERM. Thus even if the URB's completion handler always tries to resubmit, it will not succeed and the URB will become idle.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

This routine may not be used in an interrupt context (such as a bottom half or a completion handler), or when holding a spinlock, or in other situations where the caller can't schedule().

This routine should not be called by a driver after its disconnect method has returned.

void **usb_poison_urb**(struct urb * urb)
reliably kill a transfer and prevent further use of an URB

Parameters

struct urb * urb pointer to URB describing a previously submitted request, may be NULL

Description

This routine cancels an in-progress request. It is guaranteed that upon return all completion handlers will have finished and the URB will be totally idle and cannot be reused. These features make this an ideal way to stop I/O in a disconnect() callback. If the request has not already finished or been unlinked the completion handler will see urb->status == -ENOENT.

After and while the routine runs, attempts to resubmit the URB will fail with error -EPERM. Thus even if the URB's completion handler always tries to resubmit, it will not succeed and the URB will become idle.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

This routine may not be used in an interrupt context (such as a bottom half or a completion handler), or when holding a spinlock, or in other situations where the caller can't schedule().

This routine should not be called by a driver after its disconnect method has returned.

void **usb_block_urb**(struct urb * urb)
reliably prevent further use of an URB

Parameters

struct urb * urb pointer to URB to be blocked, may be NULL

Description

After the routine has run, attempts to resubmit the URB will fail with error -EPERM. Thus even if the URB's completion handler always tries to resubmit, it will not succeed and the URB will become idle.

The URB must not be deallocated while this routine is running. In particular, when a driver calls this routine, it must insure that the completion handler cannot deallocate the URB.

void **usb_kill_anchored_urbs**(struct usb_anchor * anchor)
cancel transfer requests en masse

Parameters

struct usb_anchor * anchor anchor the requests are bound to

Description

this allows all outstanding URBs to be killed starting from the back of the queue

This routine should not be called by a driver after its disconnect method has returned.

void **usb_poison_anchored_urbs**(struct usb_anchor * anchor)
cease all traffic from an anchor

Parameters

struct usb_anchor * anchor anchor the requests are bound to

Description

this allows all outstanding URBs to be poisoned starting from the back of the queue. Newly added URBs will also be poisoned

This routine should not be called by a driver after its disconnect method has returned.

void **usb_unpoison_anchored_urbs**(struct usb_anchor * anchor)
let an anchor be used successfully again

Parameters

struct usb_anchor * anchor anchor the requests are bound to

Description

Reverses the effect of usb_poison_anchored_urbs the anchor can be used normally after it returns

void **usb_unlink_anchored_urbs**(struct usb_anchor * anchor)
asynchronously cancel transfer requests en masse

Parameters

struct usb_anchor * anchor anchor the requests are bound to

Description

this allows all outstanding URBs to be unlinked starting from the back of the queue. This function is asynchronous. The unlinking is just triggered. It may happen after this function has returned.

This routine should not be called by a driver after its disconnect method has returned.

void **usb_anchor_suspend_wakeups**(struct usb_anchor * anchor)

Parameters

struct usb_anchor * anchor the anchor you want to suspend wakeups on

Description

Call this to stop the last urb being unanchored from waking up any usb_wait_anchor_empty_timeout waiters. This is used in the hcd urb give- back path to delay waking up until after the completion handler has run.

void **usb_anchor_resume_wakeups**(struct usb_anchor * anchor)

Parameters

struct usb_anchor * anchor the anchor you want to resume wakeups on

Description

Allow usb_wait_anchor_empty_timeout waiters to be woken up again, and wake up any current waiters if the anchor is empty.

int **usb_wait_anchor_empty_timeout**(struct usb_anchor * anchor, unsigned int timeout)
wait for an anchor to be unused

Parameters

struct usb_anchor * anchor the anchor you want to become unused

unsigned int timeout how long you are willing to wait in milliseconds

Description

Call this is you want to be sure all an anchor's URBs have finished

Return

Non-zero if the anchor became unused. Zero on timeout.

struct urb * **usb_get_from_anchor**(struct usb_anchor * anchor)
get an anchor's oldest urb

Parameters

struct usb_anchor * anchor the anchor whose urb you want

Description

This will take the oldest urb from an anchor, unanchor and return it

Return

The oldest urb from **anchor**, or `NULL` if **anchor** has no urbs associated with it.

void **usb_scuttle_anchored_urbs**(struct usb_anchor * anchor)
unanchor all an anchor's urbs

Parameters

struct usb_anchor * anchor the anchor whose urbs you want to unanchor

Description

use this to get rid of all an anchor's urbs

int **usb_anchor_empty**(struct usb_anchor * anchor)
is an anchor empty

Parameters

struct usb_anchor * anchor the anchor you want to query

Return

1 if the anchor has no urbs associated with it.

int **usb_control_msg**(struct usb_device * dev, unsigned int pipe, __u8 request, __u8 requesttype, __u16 value, __u16 index, void * data, __u16 size, int timeout)
Builds a control urb, sends it off and waits for completion

Parameters

struct usb_device * dev pointer to the usb device to send the message to

unsigned int pipe endpoint "pipe" to send the message to

__u8 request USB message request value

__u8 requesttype USB message request type value

__u16 value USB message value

__u16 index USB message index value

void * data pointer to the data to send

__u16 size length in bytes of the data to send

int timeout time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple control message to a specified endpoint and waits for the message to complete, or timeout.

Don't use this function from within an interrupt context. If you need an asynchronous message, or need to send a message from within interrupt context, use `usb_submit_urb()`. If a thread in your driver uses this call, make sure your disconnect() method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Return

If successful, the number of bytes transferred. Otherwise, a negative error number.

int **usb_interrupt_msg**(struct usb_device * usb_dev, unsigned int pipe, void * data, int len, int * actual_length, int timeout)

Builds an interrupt urb, sends it off and waits for completion

Parameters

struct usb_device * usb_dev pointer to the usb device to send the message to

unsigned int pipe endpoint "pipe" to send the message to

void * data pointer to the data to send

int len length in bytes of the data to send

int * actual_length pointer to a location to put the actual length transferred in bytes

int timeout time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple interrupt message to a specified endpoint and waits for the message to complete, or timeout.

Don't use this function from within an interrupt context. If you need an asynchronous message, or need to send a message from within interrupt context, use `usb_submit_urb()` If a thread in your driver uses this call, make sure your disconnect() method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Return

If successful, 0. Otherwise a negative error number. The number of actual bytes transferred will be stored in the **actual_length** parameter.

int **usb_bulk_msg**(struct usb_device * usb_dev, unsigned int pipe, void * data, int len, int * actual_length, int timeout)
Builds a bulk urb, sends it off and waits for completion

Parameters

struct usb_device * usb_dev pointer to the usb device to send the message to

unsigned int pipe endpoint "pipe" to send the message to

void * data pointer to the data to send

int len length in bytes of the data to send

int * actual_length pointer to a location to put the actual length transferred in bytes

int timeout time in msecs to wait for the message to complete before timing out (if 0 the wait is forever)

Context

!in_interrupt ()

Description

This function sends a simple bulk message to a specified endpoint and waits for the message to complete, or timeout.

Don't use this function from within an interrupt context. If you need an asynchronous message, or need to send a message from within interrupt context, use `usb_submit_urb()` If a thread in your driver uses this call, make sure your disconnect() method can wait for it to complete. Since you don't have a handle on the URB used, you can't cancel the request.

Because there is no `usb_interrupt_msg()` and no USBDEVFS_INTERRUPT ioctl, users are forced to abuse this routine by using it to submit URBs for interrupt endpoints. We will take the liberty of creating an interrupt URB (with the default interval) if the target is an interrupt endpoint.

Return

If successful, 0. Otherwise a negative error number. The number of actual bytes transferred will be stored in the **actual_length** parameter.

int **usb_sg_init**(struct usb_sg_request * io, struct usb_device * dev, unsigned pipe, unsigned period, struct scatterlist * sg, int nents, size_t length, gfp_t mem_flags)
initializes scatterlist-based bulk/interrupt I/O request

Parameters

struct usb_sg_request * io request block being initialized. until `usb_sg_wait()` returns, treat this as a pointer to an opaque block of memory,

struct usb_device * dev the usb device that will send or receive the data

unsigned pipe endpoint "pipe" used to transfer the data

unsigned period polling rate for interrupt endpoints, in frames or (for high speed endpoints) microframes; ignored for bulk

struct scatterlist * sg scatterlist entries

int nents how many entries in the scatterlist

size_t length how many bytes to send from the scatterlist, or zero to send every byte identified in the list.

gfp_t mem_flags SLAB_* flags affecting memory allocations in this call

Description

This initializes a scatter/gather request, allocating resources such as I/O mappings and urb memory (except maybe memory used by USB controller drivers).

The request must be issued using `usb_sg_wait()`, which waits for the I/O to complete (or to be canceled) and then cleans up all resources allocated by `usb_sg_init()`.

The request may be canceled with `usb_sg_cancel()`, either before or after `usb_sg_wait()` is called.

Return

Zero for success, else a negative errno value.

void **usb_sg_wait**(struct usb_sg_request * io)
synchronously execute scatter/gather request

Parameters

struct usb_sg_request * io request block handle, as initialized with `usb_sg_init()`. some fields become accessible when this call returns.

Context

!in_interrupt ()

Description

This function blocks until the specified I/O operation completes. It leverages the grouping of the related I/O requests to get good transfer rates, by queueing the requests. At higher speeds, such queuing can significantly improve USB throughput.

There are three kinds of completion for this function.

(1) success, where io->status is zero. The number of io->bytes transferred is as requested.

(2) error, where io->status is a negative errno value. The number of io->bytes transferred before the error is usually less than requested, and can be nonzero.

(3) cancellation, a type of error with status -ECONNRESET that is initiated by `usb_sg_cancel()`.

When this function returns, all memory allocated through `usb_sg_init()` or this call will have been freed. The request block parameter may still be passed to `usb_sg_cancel()`, or it may be freed. It could also be reinitialized and then reused.

Data Transfer Rates:

Bulk transfers are valid for full or high speed endpoints. The best full speed data rate is 19 packets of 64 bytes each per frame, or 1216 bytes per millisecond. The best high speed data rate is 13 packets of 512 bytes each per microframe, or 52 KBytes per millisecond.

The reason to use interrupt transfers through this API would most likely be to reserve high speed bandwidth, where up to 24 KBytes per millisecond could be transferred. That capability is less useful for low or full speed interrupt endpoints, which allow at most one packet per millisecond, of at most 8 or 64 bytes (respectively).

It is not necessary to call this function to reserve bandwidth for devices under an xHCI host controller, as the bandwidth is reserved when the configuration or interface alt setting is selected.

void **usb_sg_cancel**(struct usb_sg_request * io)
stop scatter/gather i/o issued by `usb_sg_wait()`

Parameters

struct usb_sg_request * io request block, initialized with `usb_sg_init()`

Description

This stops a request after it has been started by `usb_sg_wait()`. It can also prevents one initialized by `usb_sg_init()` from starting, so that call just frees resources allocated to the request.

int **usb_get_descriptor**(struct usb_device * dev, unsigned char type, unsigned char index, void * buf, int size)
issues a generic GET_DESCRIPTOR request

Parameters

struct usb_device * dev the device whose descriptor is being retrieved

unsigned char type the descriptor type (USB_DT_*)

unsigned char index the number of the descriptor

void * buf where to put the descriptor

int size how big is “buf” ?

Context

!in_interrupt ()

Description

Gets a USB descriptor. Convenience functions exist to simplify getting some types of descriptors. Use usb_get_string() or `usb_string()` for USB_DT_STRING. Device (USB_DT_DEVICE) and configuration descriptors (USB_DT_CONFIG) are part of the device structure. In addition to a number of USB-standard descriptors, some devices also use class-specific or vendor-specific descriptors.

This call is synchronous, and may not be used in an interrupt context.

Return

The number of bytes received on success, or else the status code returned by the underlying `usb_control_msg()` call.

int **usb_string**(struct usb_device * dev, int index, char * buf, size_t size)
returns UTF-8 version of a string descriptor

Parameters

struct usb_device * dev the device whose string descriptor is being retrieved

int index the number of the descriptor

char * buf where to put the string

size_t size how big is "buf"?

Context

!in_interrupt ()

Description

This converts the UTF-16LE encoded strings returned by devices, from usb_get_string_descriptor(), to null-terminated UTF-8 encoded ones that are more usable in most kernel contexts. Note that this function chooses strings in the first language supported by the device.

This call is synchronous, and may not be used in an interrupt context.

Return

length of the string (>= 0) or usb_control_msg status (< 0).

int **usb_get_status**(struct usb_device * dev, int recip, int type, int target, void * data)
issues a GET_STATUS call

Parameters

struct usb_device * dev the device whose status is being checked

int recip USB_RECIP_*; for device, interface, or endpoint

int type USB_STATUS_TYPE_*; for standard or PTM status types

int target zero (for device), else interface or endpoint number

void * data pointer to two bytes of bitmap data

Context

!in_interrupt ()

Description

Returns device, interface, or endpoint status. Normally only of interest to see if the device is self powered, or has enabled the remote wakeup facility; or whether a bulk or interrupt endpoint is halted (“stalled”).

Bits in these status bitmaps are set using the SET_FEATURE request, and cleared using the CLEAR_FEATURE request. The `usb_clear_halt()` function should be used to clear halt (“stall”) status.

This call is synchronous, and may not be used in an interrupt context.

Returns 0 and the status value in ***data** (in host byte order) on success, or else the status code from the underlying `usb_control_msg()` call.

int **usb_clear_halt**(struct usb_device * dev, int pipe)
tells device to clear endpoint halt/stall condition

Parameters

struct usb_device * dev device whose endpoint is halted

int pipe endpoint “pipe” being cleared

Context

!in_interrupt ()

Description

This is used to clear halt conditions for bulk and interrupt endpoints, as reported by URB completion status. Endpoints that are halted are sometimes referred to as being “stalled”. Such endpoints are unable to transmit or receive data until the halt status is cleared. Any URBs queued for such an endpoint should normally be unlinked by the driver before clearing the halt condition, as described in sections 5.7.5 and 5.8.5 of the USB 2.0 spec.

Note that control and isochronous endpoints don’t halt, although control endpoints report “protocol stall” (for unsupported requests) using the same status code used to report a true stall.

This call is synchronous, and may not be used in an interrupt context.

Return

Zero on success, or else the status code returned by the underlying `usb_control_msg()` call.

void **usb_reset_endpoint**(struct usb_device * dev, unsigned int epaddr)
Reset an endpoint’s state.

Parameters

struct usb_device * dev the device whose endpoint is to be reset

unsigned int epaddr the endpoint’s address. Endpoint number for output, endpoint number + USB_DIR_IN for input

Description

Resets any host-side endpoint state such as the toggle bit, sequence number or current window.

int **usb_set_interface**(struct usb_device * dev, int interface, int alternate)
Makes a particular alternate setting be current

Parameters

struct usb_device * dev the device whose interface is being updated

int interface the interface being updated

int alternate the setting being chosen.

Context

!in_interrupt ()

Description

This is used to enable data transfers on interfaces that may not be enabled by default. Not all devices support such configurability. Only the driver bound to an interface may change its setting.

Within any given configuration, each interface may have several alternative settings. These are often used to control levels of bandwidth consumption. For example, the default setting for a high speed interrupt endpoint may not send more than 64 bytes per microframe, while interrupt transfers of up to 3KBytes per microframe are legal. Also, isochronous endpoints may never be part of an interface' s default setting. To access such bandwidth, alternate interface settings must be made current.

Note that in the Linux USB subsystem, bandwidth associated with an endpoint in a given alternate setting is not reserved until an URB is submitted that needs that bandwidth. Some other operating systems allocate bandwidth early, when a configuration is chosen.

xHCI reserves bandwidth and configures the alternate setting in usb_hcd_alloc_bandwidth(). If it fails the original interface altsetting may be disabled. Drivers cannot rely on any particular alternate setting being in effect after a failure.

This call is synchronous, and may not be used in an interrupt context. Also, drivers must not change altsettings while urbs are scheduled for endpoints in that interface; all such urbs must first be completed (perhaps forced by unlinking).

Return

Zero on success, or else the status code returned by the underlying `usb_control_msg()` call.

int **usb_reset_configuration**(struct usb_device * dev)
lightweight device reset

Parameters

struct usb_device * dev the device whose configuration is being reset

Description

This issues a standard SET_CONFIGURATION request to the device using the current configuration. The effect is to reset most USB-related state in the device, including interface altsettings (reset to zero), endpoint halts (cleared), and endpoint

state (only for bulk and interrupt endpoints). Other usbcore state is unchanged, including bindings of usb device drivers to interfaces.

Because this affects multiple interfaces, avoid using this with composite (multi-interface) devices. Instead, the driver for each interface may use `usb_set_interface()` on the interfaces it claims. Be careful though; some devices don't support the SET_INTERFACE request, and others won't reset all the interface state (notably endpoint state). Resetting the whole configuration would affect other drivers' interfaces.

The caller must own the device lock.

Return

Zero on success, else a negative error code.

int **usb_driver_set_configuration**(struct usb_device * udev, int config)
Provide a way for drivers to change device configurations

Parameters

struct usb_device * udev the device whose configuration is being updated

int config the configuration being chosen.

Context

In process context, must be able to sleep

Description

Device interface drivers are not allowed to change device configurations. This is because changing configurations will destroy the interface the driver is bound to and create new ones; it would be like a floppy-disk driver telling the computer to replace the floppy-disk drive with a tape drive!

Still, in certain specialized circumstances the need may arise. This routine gets around the normal restrictions by using a work thread to submit the change-config request.

Return

0 if the request was successfully queued, error code otherwise. The caller has no way to know whether the queued request will eventually succeed.

int **cdc_parse_cdc_header**(struct usb_cdc_parsed_header * hdr, struct usb_interface * intf, u8 * buffer, int buflen)
parse the extra headers present in CDC devices

Parameters

struct usb_cdc_parsed_header * hdr the place to put the results of the parsing

struct usb_interface * intf the interface for which parsing is requested

u8 * buffer pointer to the extra headers to be parsed

int buflen length of the extra headers

Description

This evaluates the extra headers present in CDC devices which bind the interfaces for data and control and provide details about the capabilities of the device.

Return

number of descriptors parsed or -EINVAL if the header is contradictory beyond salvage

int **usb_register_dev**(struct usb_interface * intf, struct usb_class_driver * class_driver)
register a USB device, and ask for a minor number

Parameters

struct usb_interface * intf pointer to the usb_interface that is being registered

struct usb_class_driver * class_driver pointer to the usb_class_driver for this device

Description

This should be called by all USB drivers that use the USB major number. If CONFIG_USB_DYNAMIC_MINORS is enabled, the minor number will be dynamically allocated out of the list of available ones. If it is not enabled, the minor number will be based on the next available free minor, starting at the class_driver->minor_base.

This function also creates a usb class device in the sysfs tree.

`usb_deregister_dev()` must be called when the driver is done with the minor numbers given out by this function.

Return

-EINVAL if something bad happens with trying to register a device, and 0 on success.

void **usb_deregister_dev**(struct usb_interface * intf, struct usb_class_driver * class_driver)
deregister a USB device's dynamic minor.

Parameters

struct usb_interface * intf pointer to the usb_interface that is being deregistered

struct usb_class_driver * class_driver pointer to the usb_class_driver for this device

Description

Used in conjunction with `usb_register_dev()`. This function is called when the USB driver is finished with the minor numbers gotten from a call to `usb_register_dev()` (usually when the device is disconnected from the system.)

This function also removes the usb class device from the sysfs tree.

This should be called by all drivers that use the USB major number.

int **usb_driver_claim_interface**(struct usb_driver * driver, struct usb_interface * iface, void * priv)
bind a driver to an interface

Parameters

struct usb_driver * driver the driver to be bound

struct usb_interface * iface the interface to which it will be bound; must be in the usb device's active configuration

void * priv driver data associated with that interface

Description

This is used by usb device drivers that need to claim more than one interface on a device when probing (audio and acm are current examples). No device driver should directly modify internal usb_interface or usb_device structure members.

Few drivers should need to use this routine, since the most natural way to bind to an interface is to return the private data from the driver's probe() method.

Callers must own the device lock, so driver probe() entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

Return

0 on success.

void **usb_driver_release_interface**(struct usb_driver * driver, struct usb_interface * iface)
unbind a driver from an interface

Parameters

struct usb_driver * driver the driver to be unbound

struct usb_interface * iface the interface from which it will be unbound

Description

This can be used by drivers to release an interface without waiting for their disconnect() methods to be called. In typical cases this also causes the driver disconnect() method to be called.

This call is synchronous, and may not be used in an interrupt context. Callers must own the device lock, so driver disconnect() entries don't need extra locking, but other call contexts may need to explicitly claim that lock.

const struct usb_device_id * **usb_match_id**(struct usb_interface * interface, const struct usb_device_id * id)
find first usb_device_id matching device or interface

Parameters

struct usb_interface * interface the interface of interest

const struct usb_device_id * id array of usb_device_id structures, terminated by zero entry

Description

usb_match_id searches an array of usb_device_id's and returns the first one matching the device or interface, or null. This is used when binding (or rebinding) a driver to an interface. Most USB device drivers will use this indirectly, through the usb core, but some layered driver frameworks use it directly. These device tables are exported with MODULE_DEVICE_TABLE, through modutils, to support the driver loading functionality of USB hotplugging.

What Matches:

The “match_flags” element in a usb_device_id controls which members are used. If the corresponding bit is set, the value in the device_id must match its corresponding member in the device or interface descriptor, or else the device_id does not match.

“driver_info” is normally used only by device drivers, but you can create a wildcard “matches anything” usb_device_id as a driver’s “modules.usbmap” entry if you provide an id with only a nonzero “driver_info” field. If you do this, the USB device driver’s probe() routine should use additional intelligence to decide whether to bind to the specified interface.

What Makes Good usb_device_id Tables:

The match algorithm is very simple, so that intelligence in driver selection must come from smart driver id records. Unless you have good reasons to use another selection policy, provide match elements only in related groups, and order match specifiers from specific to general. Use the macros provided for that purpose if you can.

The most specific match specifiers use device descriptor data. These are commonly used with product-specific matches; the USB_DEVICE macro lets you provide vendor and product IDs, and you can also match against ranges of product revisions. These are widely used for devices with application or vendor specific bDeviceClass values.

Matches based on device class/subclass/protocol specifications are slightly more general; use the USB_DEVICE_INFO macro, or its siblings. These are used with single-function devices where bDeviceClass doesn’t specify that each interface has its own class.

Matches based on interface class/subclass/protocol are the most general; they let drivers bind to any interface on a multiple-function device. Use the USB_INTERFACE_INFO macro, or its siblings, to match class-per-interface style devices (as recorded in bInterfaceClass).

Note that an entry created by USB_INTERFACE_INFO won’t match any interface if the device class is set to Vendor-Specific. This is deliberate; according to the USB spec the meanings of the interface class/subclass/protocol for these devices are also vendor-specific, and hence matching against a standard product class wouldn’t work anyway. If you really want to use an interface-based match for such a device, create a match record that also specifies the vendor ID. (Unforunately there isn’t a standard macro for creating records like this.)

Within those groups, remember that not all combinations are meaningful. For example, don’t give a product version range without vendor and product IDs; or specify a protocol without its associated class and subclass.

Return

The first matching usb_device_id, or `NULL`.

int **usb_register_device_driver**(struct usb_device_driver * new_udriver,
struct module * owner)
register a USB device (not interface) driver

Parameters

struct usb_device_driver * new_udriver USB operations for the device driver

struct module * owner module owner of this driver.

Description

Registers a USB device driver with the USB core. The list of unattached devices will be rescanned whenever a new driver is added, allowing the new driver to attach to any recognized devices.

Return

A negative error code on failure and 0 on success.

void **usb_deregister_device_driver**(struct usb_device_driver * udriver)
unregister a USB device (not interface) driver

Parameters

struct usb_device_driver * udriver USB operations of the device driver to unregister

Context

must be able to sleep

Description

Unlinks the specified driver from the internal USB driver list.

int **usb_register_driver**(struct usb_driver * new_driver, struct module * owner, const char * mod_name)
register a USB interface driver

Parameters

struct usb_driver * new_driver USB operations for the interface driver

struct module * owner module owner of this driver.

const char * mod_name module name string

Description

Registers a USB interface driver with the USB core. The list of unattached interfaces will be rescanned whenever a new driver is added, allowing the new driver to attach to any recognized interfaces.

Return

A negative error code on failure and 0 on success.

NOTE

if you want your driver to use the USB major number, you must call `usb_register_dev()` to enable that functionality. This function no longer takes care of that.

void **usb_deregister**(struct usb_driver * driver)
unregister a USB interface driver

Parameters

struct usb_driver * driver USB operations of the interface driver to unregister

Context

must be able to sleep

Description

Unlinks the specified driver from the internal USB driver list.

NOTE

If you called `usb_register_dev()`, you still need to call `usb_deregister_dev()` to clean up your driver's allocated minor numbers, this * call will no longer do it for you.

void **usb_enable_autosuspend**(struct usb_device * udev)
allow a USB device to be autosuspended

Parameters

struct usb_device * udev the USB device which may be autosuspended

Description

This routine allows **udev** to be autosuspended. An autosuspend won't take place until the autosuspend_delay has elapsed and all the other necessary conditions are satisfied.

The caller must hold **udev**'s device lock.

void **usb_disable_autosuspend**(struct usb_device * udev)
prevent a USB device from being autosuspended

Parameters

struct usb_device * udev the USB device which may not be autosuspended

Description

This routine prevents **udev** from being autosuspended and wakes it up if it is already autosuspended.

The caller must hold **udev**'s device lock.

void **usb_autopm_put_interface**(struct usb_interface * intf)
decrement a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be decremented

Description

This routine should be called by an interface driver when it is finished using **intf** and wants to allow it to autosuspend. A typical example would be a character-device driver when its device file is closed.

The routine decrements **intf**'s usage counter. When the counter reaches 0, a delayed autosuspend request for **intf**'s device is attempted. The attempt may fail (see autosuspend_check()).

This routine can run only in process context.

void **usb_autopm_put_interface_async**(struct usb_interface * intf)
decrement a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be decremented

Description

This routine does much the same thing as `usb_autopm_put_interface()`: It decrements **intf**'s usage counter and schedules a delayed autosuspend request if the counter is <= 0. The difference is that it does not perform any synchronization; callers should hold a private lock and handle all synchronization issues themselves.

Typically a driver would call this routine during an URB's completion handler, if no more URBs were pending.

This routine can run in atomic context.

void **usb_autopm_put_interface_no_suspend**(struct usb_interface * intf)
decrement a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be decremented

Description

This routine decrements **intf**'s usage counter but does not carry out an autosuspend.

This routine can run in atomic context.

int **usb_autopm_get_interface**(struct usb_interface * intf)
increment a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be incremented

Description

This routine should be called by an interface driver when it wants to use **intf** and needs to guarantee that it is not suspended. In addition, the routine prevents **intf** from being autosuspended subsequently. (Note that this will not prevent suspend events originating in the PM core.) This prevention will persist until `usb_autopm_put_interface()` is called or **intf** is unbound. A typical example would be a character-device driver when its device file is opened.

intf's usage counter is incremented to prevent subsequent autosuspends. However if the autoresume fails then the counter is re-decremented.

This routine can run only in process context.

Return

0 on success.

int **usb_autopm_get_interface_async**(struct usb_interface * intf)
increment a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be incremented

Description

This routine does much the same thing as `usb_autopm_get_interface()`: It increments **intf**'s usage counter and queues an autoresume request if the device is suspended. The differences are that it does not perform any synchronization (callers should hold a private lock and handle all synchronization issues themselves), and it does not autoresume the device directly (it only queues a request). After a successful call, the device may not yet be resumed.

This routine can run in atomic context.

Return

0 on success. A negative error code otherwise.

void **usb_autopm_get_interface_no_resume**(struct usb_interface * intf)
increment a USB interface's PM-usage counter

Parameters

struct usb_interface * intf the usb_interface whose counter should be incremented

Description

This routine increments **intf**'s usage counter but does not carry out an autoresume.

This routine can run in atomic context.

int **usb_find_common_endpoints**(struct usb_host_interface * alt, struct usb_endpoint_descriptor ** bulk_in, struct usb_endpoint_descriptor ** bulk_out, struct usb_endpoint_descriptor ** int_in, struct usb_endpoint_descriptor ** int_out)

- look up common endpoint descriptors

Parameters

struct usb_host_interface * alt alternate setting to search

struct usb_endpoint_descriptor ** bulk_in pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** bulk_out pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** int_in pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** int_out pointer to descriptor pointer, or NULL

Description

Search the alternate setting's endpoint descriptors for the first bulk-in, bulk-out, interrupt-in and interrupt-out endpoints and return them in the provided pointers (unless they are NULL).

If a requested endpoint is not found, the corresponding pointer is set to NULL.

Return

Zero if all requested descriptors were found, or -ENXIO otherwise.

int **usb_find_common_endpoints_reverse**(struct usb_host_interface * alt, struct usb_endpoint_descriptor ** bulk_in, struct usb_endpoint_descriptor ** bulk_out, struct usb_endpoint_descriptor ** int_in, struct usb_endpoint_descriptor ** int_out)

- look up common endpoint descriptors

Parameters

struct usb_host_interface * alt alternate setting to search

struct usb_endpoint_descriptor ** bulk_in pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** bulk_out pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** int_in pointer to descriptor pointer, or NULL

struct usb_endpoint_descriptor ** int_out pointer to descriptor pointer, or NULL

Description

Search the alternate setting's endpoint descriptors for the last bulk-in, bulk-out, interrupt-in and interrupt-out endpoints and return them in the provided pointers (unless they are NULL).

If a requested endpoint is not found, the corresponding pointer is set to NULL.

Return

Zero if all requested descriptors were found, or -ENXIO otherwise.

struct usb_host_interface * **usb_find_alt_setting**(struct usb_host_config * config, unsigned int iface_num, unsigned int alt_num)

Given a configuration, find the alternate setting for the given interface.

Parameters

struct usb_host_config * config the configuration to search (not necessarily the current config).

unsigned int iface_num interface number to search in

unsigned int alt_num alternate interface setting number to search for.

Description

Search the configuration's interface cache for the given alt setting.

Return

The alternate setting, if found. NULL otherwise.

struct usb_interface * **usb_ifnum_to_if**(const struct usb_device * dev, unsigned ifnum)
get the interface object with a given interface number

Parameters

const struct usb_device * dev the device whose current configuration is considered

unsigned ifnum the desired interface

Description

This walks the device descriptor for the currently active configuration to find the interface object with the particular interface number.

Note that configuration descriptors are not required to assign interface numbers sequentially, so that it would be incorrect to assume that the first interface in that descriptor corresponds to interface zero. This routine helps device drivers avoid such mistakes. However, you should make sure that you do the right thing with any alternate settings available for this interfaces.

Don't call this function unless you are bound to one of the interfaces on this device or you have locked the device!

Return

A pointer to the interface that has **ifnum** as interface number, if found. NULL otherwise.

struct usb_host_interface * **usb_altnum_to_altsetting**(const struct usb_interface * intf, unsigned int altnum)
get the altsetting structure with a given alternate setting number.

Parameters

const struct usb_interface * intf the interface containing the altsetting in question

unsigned int altnum the desired alternate setting number

Description

This searches the altsetting array of the specified interface for an entry with the correct bAlternateSetting value.

Note that altsettings need not be stored sequentially by number, so it would be incorrect to assume that the first altsetting entry in the array corresponds to altsetting zero. This routine helps device drivers avoid such mistakes.

Don't call this function unless you are bound to the intf interface or you have locked the device!

Return

A pointer to the entry of the altsetting array of **intf** that has **altnum** as the alternate setting number. `NULL` if not found.

struct usb_interface * **usb_find_interface**(struct usb_driver * drv, int minor)
find usb_interface pointer for driver and device

Parameters

struct usb_driver * drv the driver whose current configuration is considered

int minor the minor number of the desired device

Description

This walks the bus device list and returns a pointer to the interface with the matching minor and driver. Note, this only works for devices that share the USB major number.

Return

A pointer to the interface with the matching major and **minor**.

int **usb_for_each_dev**(void * data, int (*fn)(struct usb_device *, void *))
iterate over all USB devices in the system

Parameters

void * data data pointer that will be handed to the callback function

int (*)(struct usb_device *, void *) fn callback function to be called for each USB device

Description

Iterate over all USB devices and call **fn** for each, passing it **data**. If it returns anything other than 0, we break the iteration prematurely and return that value.

struct usb_device * **usb_alloc_dev**(struct usb_device * parent, struct usb_bus * bus, unsigned port1)
usb device constructor (usbcore-internal)

Parameters

struct usb_device * parent hub to which device is connected; null to allocate a root hub

struct usb_bus * bus bus used to access the device

unsigned port1 one-based index of port; ignored for root hubs

Context

!in_interrupt()

Description

Only hub drivers (including virtual root hub drivers for host controllers) should ever call this.

This call may not be used in a non-sleeping context.

Return

On success, a pointer to the allocated usb device. NULL on failure.

struct usb_device * **usb_get_dev**(struct usb_device * dev)
increments the reference count of the usb device structure

Parameters

struct usb_device * dev the device being referenced

Description

Each live reference to a device should be refcounted.

Drivers for USB interfaces should normally record such references in their probe() methods, when they bind to an interface, and release them by calling usb_put_dev(), in their disconnect() methods.

Return

A pointer to the device with the incremented reference counter.

void **usb_put_dev**(struct usb_device * dev)
release a use of the usb device structure

Parameters

struct usb_device * dev device that's been disconnected

Description

Must be called when a user of a device is finished with it. When the last user of the device calls this function, the memory of the device is freed.

struct usb_interface * **usb_get_intf**(struct usb_interface * intf)
increments the reference count of the usb interface structure

Parameters

struct usb_interface * intf the interface being referenced

Description

Each live reference to a interface must be refcounted.

Drivers for USB interfaces should normally record such references in their probe() methods, when they bind to an interface, and release them by calling usb_put_intf(), in their disconnect() methods.

Return

A pointer to the interface with the incremented reference counter.

void **usb_put_intf**(struct usb_interface * intf)
release a use of the usb interface structure

Parameters

struct usb_interface * intf interface that's been decremented

Description

Must be called when a user of an interface is finished with it. When the last user of the interface calls this function, the memory of the interface is freed.

int **usb_lock_device_for_reset**(struct usb_device * udev, const struct usb_interface * iface)
cautiously acquire the lock for a usb device structure

Parameters

struct usb_device * udev device that's being locked

const struct usb_interface * iface interface bound to the driver making the request (optional)

Description

Attempts to acquire the device lock, but fails if the device is NOTATTACHED or SUSPENDED, or if iface is specified and the interface is neither BINDING nor BOUND. Rather than sleeping to wait for the lock, the routine polls repeatedly. This is to prevent deadlock with disconnect; in some drivers (such as usb-storage) the disconnect() or suspend() method will block waiting for a device reset to complete.

Return

A negative error code for failure, otherwise 0.

int **usb_get_current_frame_number**(struct usb_device * dev)
return current bus frame number

Parameters

struct usb_device * dev the device whose bus is being queried

Return

The current frame number for the USB host controller used with the given USB device. This can be used when scheduling isochronous requests.

Note

Different kinds of host controller have different "scheduling horizons". While one type might support scheduling only 32 frames into the future, others could support scheduling up to 1024 frames into the future.

void * **usb_alloc_coherent**(struct usb_device * dev, size_t size, gfp_t mem_flags, dma_addr_t * dma)
allocate dma-consistent buffer for URB_NO_xxx_DMA_MAP

Parameters

struct usb_device * dev device the buffer will be used with

size_t size requested buffer size

gfp_t mem_flags affect whether allocation may block

dma_addr_t * dma used to return DMA address of buffer

Return

Either null (indicating no buffer could be allocated), or the cpu-space pointer to a buffer that may be used to perform DMA to the specified device. Such cpu-space buffers are returned along with the DMA address (through the pointer provided).

Note

These buffers are used with URB_NO_xxx_DMA_MAP set in urb->transfer_flags to avoid behaviors like using "DMA bounce buffers", or thrashing IOMMU hardware during URB completion/resubmit. The implementation varies between platforms, depending on details of how DMA will work to this device. Using these buffers also eliminates cacheline sharing problems on architectures where CPU caches are not DMA-coherent. On systems without bus-snooping caches, these buffers are uncached.

Description

When the buffer is no longer used, free it with `usb_free_coherent()`.

void **usb_free_coherent**(struct usb_device * dev, size_t size, void * addr, dma_addr_t dma)
free memory allocated with `usb_alloc_coherent()`

Parameters

struct usb_device * dev device the buffer was used with

size_t size requested buffer size

void * addr CPU address of buffer

dma_addr_t dma DMA address of buffer

Description

This reclaims an I/O buffer, letting it be reused. The memory must have been allocated using `usb_alloc_coherent()`, and the parameters must match those provided in that allocation request.

int **usb_hub_clear_tt_buffer**(struct urb * urb)
clear control/bulk TT state in high speed hub

Parameters

struct urb * urb an URB associated with the failed or incomplete split transaction

Description

High speed HCDs use this to tell the hub driver that some split control or bulk transaction failed in a way that requires clearing internal state of a transaction translator. This is normally detected (and reported) from interrupt context.

It may not be possible for that hub to handle additional full (or low) speed transactions until that state is fully cleared out.

Return

0 if successful. A negative error code otherwise.

void **usb_set_device_state**(struct usb_device * udev, enum usb_device_state new_state)
change a device's current state (usbcore, hcds)

Parameters

struct usb_device * udev pointer to device whose state should be changed

enum usb_device_state new_state new state value to be stored

Description

udev->state is _not_ fully protected by the device lock. Although most transitions are made only while holding the lock, the state can can change to USB_STATE_NOTATTACHED at almost any time. This is so that devices can be marked as disconnected as soon as possible, without having to wait for any semaphores to be released. As a result, all changes to any device's state must be protected by the device_state_lock spinlock.

Once a device has been added to the device tree, all changes to its state should be made using this routine. The state should _not_ be set directly.

If udev->state is already USB_STATE_NOTATTACHED then no change is made. Otherwise udev->state is set to new_state, and if new_state is USB_STATE_NOTATTACHED then all of udev's descendants' states are also set to USB_STATE_NOTATTACHED.

void **usb_root_hub_lost_power**(struct usb_device * rhdev)
called by HCD if the root hub lost Vbus power

Parameters

struct usb_device * rhdev struct usb_device for the root hub

Description

The USB host controller driver calls this function when its root hub is resumed and Vbus power has been interrupted or the controller has been reset. The routine marks **rhdev** as having lost power. When the hub driver is resumed it will take notice and carry out power-session recovery for all the "USB-PERSIST"-enabled child devices; the others will be disconnected.

int **usb_reset_device**(struct usb_device * udev)
warn interface drivers and perform a USB port reset

Parameters

struct usb_device * udev device to reset (not in NOTATTACHED state)

Description

Warns all drivers bound to registered interfaces (using their pre_reset method), performs the port reset, and then lets the drivers know that the reset is over (using their post_reset method).

If an interface is currently being probed or disconnected, we assume its driver knows how to handle resets. For all other interfaces, if the driver doesn't have pre_reset and post_reset methods then we attempt to unbind it and rebind afterward.

Return

The same as for usb_reset_and_verify_device().

Note

The caller must own the device lock. For example, it's safe to use this from a driver probe() routine after downloading new firmware. For calls that might not occur during probe(), drivers should lock the device using `usb_lock_device_for_reset()`.

void **usb_queue_reset_device**(struct usb_interface * iface)
Reset a USB device from an atomic context

Parameters

struct usb_interface * iface USB interface belonging to the device to reset

Description

This function can be used to reset a USB device from an atomic context, where `usb_reset_device()` won't work (as it blocks).

Doing a reset via this method is functionally equivalent to calling `usb_reset_device()`, except for the fact that it is delayed to a workqueue. This means that any drivers bound to other interfaces might be unbound, as well as users from usbfs in user space.

Corner cases:

- Scheduling two resets at the same time from two different drivers attached to two different interfaces of the same device is possible; depending on how the driver attached to each interface handles ->pre_reset(), the second reset might happen or not.
- If the reset is delayed so long that the interface is unbound from its driver, the reset will be skipped.
- This function can be called during .probe(). It can also be called during .disconnect(), but doing so is pointless because the reset will not occur. If you really want to reset the device during .disconnect(), call `usb_reset_device()` directly – but watch out for nested unbinding issues!

struct usb_device * **usb_hub_find_child**(struct usb_device * hdev,
int port1)
Get the pointer of child device attached to the port which is specified by **port1**.

Parameters

struct usb_device * hdev USB device belonging to the usb hub

int port1 port num to indicate which port the child device is attached to.

Description

USB drivers call this function to get hub's child device pointer.

Return

`NULL` if input param is invalid and child's usb_device pointer if non-NULL.

20.1.6 Host Controller APIs

These APIs are only for use by host controller drivers, most of which implement standard register interfaces such as XHCI, EHCI, OHCI, or UHCI. UHCI was one of the first interfaces, designed by Intel and also used by VIA; it doesn't do much in hardware. OHCI was designed later, to have the hardware do more work (bigger transfers, tracking protocol state, and so on). EHCI was designed with USB 2.0; its design has features that resemble OHCI (hardware does much more work) as well as UHCI (some parts of ISO support, TD list processing). XHCI was designed with USB 3.0. It continues to shift support for functionality into hardware.

There are host controllers other than the "big three", although most PCI based controllers (and a few non-PCI based ones) use one of those interfaces. Not all host controllers use DMA; some use PIO, and there is also a simulator and a virtual host controller to pipe USB over the network.

The same basic APIs are available to drivers for all those controllers. For historical reasons they are in two layers: `struct usb_bus` is a rather thin layer that became available in the 2.2 kernels, while `struct usb_hcd` is a more featureful layer that lets HCDs share common code, to shrink driver size and significantly reduce hcd-specific behaviors.

long **usb_calc_bus_time**(int speed, int is_input, int isoc, int bytecount)
approximate periodic transaction time in nanoseconds

Parameters

int speed from dev->speed; USB_SPEED_{LOW,FULL,HIGH}

int is_input true iff the transaction sends data to the host

int isoc true for isochronous transactions, false for interrupt ones

int bytecount how many bytes in the transaction.

Return

Approximate bus time in nanoseconds for a periodic transaction.

Note

See USB 2.0 spec section 5.11.3; only periodic transfers need to be scheduled in software, this function is only used for such scheduling.

int **usb_hcd_link_urb_to_ep**(struct usb_hcd * hcd, struct urb * urb)
add an URB to its endpoint queue

Parameters

struct usb_hcd * hcd host controller to which **urb** was submitted

struct urb * urb URB being submitted

Description

Host controller drivers should call this routine in their enqueue() method. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for URB submission, as well as for endpoint shutdown and for usb_kill_urb.

Return

0 for no error, otherwise a negative error code (in which case the enqueue() method must fail). If no error occurs but enqueue() fails anyway, it must call `usb_hcd_unlink_urb_from_ep()` before releasing the private spinlock and returning.

int **usb_hcd_check_unlink_urb**(struct usb_hcd * hcd, struct urb * urb, int status)
check whether an URB may be unlinked

Parameters

struct usb_hcd * hcd host controller to which **urb** was submitted

struct urb * urb URB being checked for unlinkability

int status error code to store in **urb** if the unlink succeeds

Description

Host controller drivers should call this routine in their dequeue() method. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for making sure than an unlink is valid.

Return

0 for no error, otherwise a negative error code (in which case the dequeue() method must fail). The possible error codes are:

-EIDRM: urb was not submitted or has already completed. The completion function may not have been called yet.

-EBUSY: **urb** has already been unlinked.

void **usb_hcd_unlink_urb_from_ep**(struct usb_hcd * hcd, struct urb * urb)
remove an URB from its endpoint queue

Parameters

struct usb_hcd * hcd host controller to which **urb** was submitted

struct urb * urb URB being unlinked

Description

Host controller drivers should call this routine before calling `usb_hcd_giveback_urb()`. The HCD's private spinlock must be held and interrupts must be disabled. The actions carried out here are required for URB completion.

void **usb_hcd_giveback_urb**(struct usb_hcd * hcd, struct urb * urb, int status)
return URB from HCD to device driver

Parameters

struct usb_hcd * hcd host controller returning the URB

struct urb * urb urb being returned to the USB device driver.

int status completion status code for the URB.

Context

in_interrupt()

Description

This hands the URB from HCD to its USB device driver, using its completion function. The HCD has freed all per-urb resources (and is done using urb->hcpriv). It also released all HCD locks; the device driver won't cause problems if it frees, modifies, or resubmits this URB.

If **urb** was unlinked, the value of **status** will be overridden by **urb->unlinked**. Erroneous short transfers are detected in case the HCD hasn't checked for them.

int **usb_alloc_streams**(struct usb_interface * interface, struct usb_host_endpoint ** eps, unsigned int num_eps, unsigned int num_streams, gfp_t mem_flags)
allocate bulk endpoint stream IDs.

Parameters

struct usb_interface * interface alternate setting that includes all endpoints.

struct usb_host_endpoint ** eps array of endpoints that need streams.

unsigned int num_eps number of endpoints in the array.

unsigned int num_streams number of streams to allocate.

gfp_t mem_flags flags hcd should use to allocate memory.

Description

Sets up a group of bulk endpoints to have **num_streams** stream IDs available. Drivers may queue multiple transfers to different stream IDs, which may complete in a different order than they were queued.

Return

On success, the number of allocated streams. On failure, a negative error code.

int **usb_free_streams**(struct usb_interface * interface, struct usb_host_endpoint ** eps, unsigned int num_eps, gfp_t mem_flags)
free bulk endpoint stream IDs.

Parameters

struct usb_interface * interface alternate setting that includes all endpoints.

struct usb_host_endpoint ** eps array of endpoints to remove streams from.

unsigned int num_eps number of endpoints in the array.

gfp_t mem_flags flags hcd should use to allocate memory.

Description

Reverts a group of bulk endpoints back to not using stream IDs. Can fail if we are given bad arguments, or HCD is broken.

Return

0 on success. On failure, a negative error code.

void **usb_hcd_resume_root_hub**(struct usb_hcd * hcd)
called by HCD to resume its root hub

Parameters

struct usb_hcd * hcd host controller for this root hub

Description

The USB host controller calls this function when its root hub is suspended (with the remote wakeup feature enabled) and a remote wakeup request is received. The routine submits a workqueue request to resume the root hub (that is, manage its downstream ports again).

int **usb_bus_start_enum**(struct usb_bus * bus, unsigned port_num)
start immediate enumeration (for OTG)

Parameters

struct usb_bus * bus the bus (must use hcd framework)

unsigned port_num 1-based number of port; usually bus->otg_port

Context

in_interrupt()

Description

Starts enumeration, with an immediate reset followed later by hub_wq identifying and possibly configuring the device. This is needed by OTG controller drivers, where it helps meet HNP protocol timing requirements for starting a port reset.

Return

0 if successful.

irqreturn_t **usb_hcd_irq**(int irq, void * __hcd)
hook IRQs to HCD framework (bus glue)

Parameters

int irq the IRQ being raised

void * __hcd pointer to the HCD whose IRQ is being signaled

Description

If the controller isn't HALTed, calls the driver's irq handler. Checks whether the controller is now dead.

Return

`IRQ_HANDLED` if the IRQ was handled. `IRQ_NONE` otherwise.

void **usb_hc_died**(struct usb_hcd * hcd)
report abnormal shutdown of a host controller (bus glue)

Parameters

struct usb_hcd * hcd pointer to the HCD representing the controller

Description

This is called by bus glue to report a USB host controller that died while operations may still have been pending. It's called automatically by the PCI glue, so only glue for non-PCI busses should need to call it.

Only call this function with the primary HCD.

struct usb_hcd * **usb_create_shared_hcd**(const struct hc_driver * driver, struct device * dev, const char * bus_name, struct usb_hcd * primary_hcd)
create and initialize an HCD structure

Parameters

const struct hc_driver * driver HC driver that will use this hcd

struct device * dev device for this HC, stored in hcd->self.controller

const char * bus_name value to store in hcd->self.bus_name

struct usb_hcd * primary_hcd a pointer to the usb_hcd structure that is sharing the PCI device. Only allocate certain resources for the primary HCD

Context

!in_interrupt()

Description

Allocate a struct usb_hcd, with extra space at the end for the HC driver's private data. Initialize the generic members of the hcd structure.

Return

On success, a pointer to the created and initialized HCD structure. On failure (e.g. if memory is unavailable), `NULL`.

struct usb_hcd * **usb_create_hcd**(const struct hc_driver * driver, struct device * dev, const char * bus_name)
create and initialize an HCD structure

Parameters

const struct hc_driver * driver HC driver that will use this hcd

struct device * dev device for this HC, stored in hcd->self.controller

const char * bus_name value to store in hcd->self.bus_name

Context

!in_interrupt()

Description

Allocate a struct usb_hcd, with extra space at the end for the HC driver's private data. Initialize the generic members of the hcd structure.

Return

On success, a pointer to the created and initialized HCD structure. On failure (e.g. if memory is unavailable), `NULL`.

int **usb_add_hcd**(struct usb_hcd * hcd, unsigned int irqnum, unsigned long irqflags)
finish generic HCD structure initialization and register

Parameters

struct usb_hcd * hcd the usb_hcd structure to initialize

unsigned int irqnum Interrupt line to allocate

unsigned long irqflags Interrupt type flags

Description

Finish the remaining parts of generic HCD initialization: allocate the buffers of consistent memory, register the bus, request the IRQ line, and call the driver's reset() and start() routines.

void **usb_remove_hcd**(struct usb_hcd * hcd)
shutdown processing for generic HCDs

Parameters

struct usb_hcd * hcd the usb_hcd structure to remove

Context

!in_interrupt()

Description

Disconnects the root hub, then reverses the effects of `usb_add_hcd()`, invoking the HCD's stop() method.

int **usb_hcd_pci_probe**(struct pci_dev * dev, const struct pci_device_id * id, const struct hc_driver * driver)
initialize PCI-based HCDs

Parameters

struct pci_dev * dev USB Host Controller being probed

const struct pci_device_id * id pci hotplug id connecting controller to HCD framework

const struct hc_driver * driver USB HC driver handle

Context

!in_interrupt()

Description

Allocates basic PCI resources for this USB host controller, and then invokes the start() method for the HCD associated with it through the hotplug entry's driver_data.

Store this function in the HCD's struct pci_driver as probe().

Return

0 if successful.

void **usb_hcd_pci_remove**(struct pci_dev * dev)
shutdown processing for PCI-based HCDs

Parameters

struct pci_dev * dev USB Host Controller being removed

Context

!in_interrupt()

Description

Reverses the effect of `usb_hcd_pci_probe()`, first invoking the HCD's stop() method. It is always called from a thread context, normally "rmmod", "apmd", or something similar.

Store this function in the HCD's struct pci_driver as remove().

void **usb_hcd_pci_shutdown**(struct pci_dev * dev)
shutdown host controller

Parameters

struct pci_dev * dev USB Host Controller being shutdown

int **hcd_buffer_create**(struct usb_hcd * hcd)
initialize buffer pools

Parameters

struct usb_hcd * hcd the bus whose buffer pools are to be initialized

Context

!in_interrupt()

Description

Call this as part of initializing a host controller that uses the dma memory allocators. It initializes some pools of dma-coherent memory that will be shared by all drivers using that controller.

Call `hcd_buffer_destroy()` to clean up after using those pools.

Return

0 if successful. A negative errno value otherwise.

void **hcd_buffer_destroy**(struct usb_hcd * hcd)
deallocate buffer pools

Parameters

struct usb_hcd * hcd the bus whose buffer pools are to be destroyed

Context

!in_interrupt()

Description

This frees the buffer pools created by `hcd_buffer_create()`.

20.1.7 The USB character device nodes

This chapter presents the Linux character device nodes. You may prefer to avoid writing new kernel code for your USB driver. User mode device drivers are usually packaged as applications or libraries, and may use character devices through some programming library that wraps it. Such libraries include:

- libusb for C/C++, and
- jUSB for Java.

Some old information about it can be seen at the "USB Device Filesystem" section of the USB Guide. The latest copy of the USB Guide can be found at http://www.linux-usb.org/

Note:

- They were used to be implemented via usbfs, but this is not part of the sysfs debug interface.
- This particular documentation is incomplete, especially with respect to the asynchronous mode. As of kernel 2.5.66 the code and this (new) documentation need to be cross-reviewed.

What files are in "devtmpfs"?

Conventionally mounted at `/dev/bus/usb/`, usbfs features include:

- `/dev/bus/usb/BBB/DDD` ⋯magic files exposing the each device's configuration descriptors, and supporting a series of ioctls for making device requests, including I/O to devices. (Purely for access by programs.)

Each bus is given a number (`BBB`) based on when it was enumerated; within each bus, each device is given a similar number (`DDD`). Those `BBB/DDD` paths are not "stable" identifiers; expect them to change even if you always leave the devices plugged in to the same hub port. Don't even think of saving these in application configuration files. Stable identifiers are available, for user mode applications that want to use them. HID and networking devices expose these stable IDs, so that for example you can be sure that you told the right UPS to power down its second server. Pleast note that it doesn't (yet) expose those IDs.

/dev/bus/usb/BBB/DDD

Use these files in one of these basic ways:

- They can be read, producing first the device descriptor (18 bytes) and then the descriptors for the current configuration. See the USB 2.0 spec for details about those binary data formats. You'll need to convert most multibyte values from little endian format to your native host byte order, although a few of the fields in the device descriptor (both of the BCD-encoded fields, and the vendor and product IDs) will be byteswapped for you. Note that configuration descriptors include descriptors for interfaces, altsettings, endpoints, and maybe additional class descriptors.

- Perform USB operations using ioctl() requests to make endpoint I/O requests (synchronously or asynchronously) or manage the device. These requests need the CAP_SYS_RAWIO capability, as well as filesystem access permissions. Only one ioctl request can be made on one of these device files at a time. This means that if you are synchronously reading an endpoint from one thread, you won't be able to write to a different endpoint from another thread until the read completes. This works for half duplex protocols, but otherwise you'd use asynchronous i/o requests.

Each connected USB device has one file. The BBB indicates the bus number. The DDD indicates the device address on that bus. Both of these numbers are assigned sequentially, and can be reused, so you can't rely on them for stable access to devices. For example, it's relatively common for devices to re-enumerate while they are still connected (perhaps someone jostled their power supply, hub, or USB cable), so a device might be 002/027 when you first connect it and 002/048 sometime later.

These files can be read as binary data. The binary data consists of first the device descriptor, then the descriptors for each configuration of the device. Multi-byte fields in the device descriptor are converted to host endianness by the kernel. The configuration descriptors are in bus endian format! The configuration descriptor are wTotalLength bytes apart. If a device returns less configuration descriptor data than indicated by wTotalLength there will be a hole in the file for the missing bytes. This information is also shown in text form by the /sys/kernel/debug/usb/devices file, described later.

These files may also be used to write user-level drivers for the USB devices. You would open the /dev/bus/usb/BBB/DDD file read/write, read its descriptors to make sure it's the device you expect, and then bind to an interface (or perhaps several) using an ioctl call. You would issue more ioctls to the device to communicate to it using control, bulk, or other kinds of USB transfers. The IOCTLs are listed in the <linux/usbdevice_fs.h> file, and at this writing the source code (linux/drivers/usb/core/devio.c) is the primary reference for how to access devices through those files.

Note that since by default these BBB/DDD files are writable only by root, only root can write such user mode drivers. You can selectively grant read/write permissions to other users by using chmod. Also, usbfs mount options such as devmode=0666 may be helpful.

Life Cycle of User Mode Drivers

Such a driver first needs to find a device file for a device it knows how to handle. Maybe it was told about it because a /sbin/hotplug event handling agent chose that driver to handle the new device. Or maybe it's an application that scans all the /dev/bus/usb device files, and ignores most devices. In either case, it should read() all the descriptors from the device file, and check them against what it knows how to handle. It might just reject everything except a particular vendor and product ID, or need a more complex policy.

Never assume there will only be one such device on the system at a time! If your code can't handle more than one device at a time, at least detect when there's more than one, and have your users choose which device to use.

Once your user mode driver knows what device to use, it interacts with it in either of two styles. The simple style is to make only control requests; some devices don't need more complex interactions than those. (An example might be software using vendor-specific control requests for some initialization or configuration tasks, with a kernel driver for the rest.)

More likely, you need a more complex style driver: one using non-control endpoints, reading or writing data and claiming exclusive use of an interface. Bulk transfers are easiest to use, but only their sibling interrupt transfers work with low speed devices. Both interrupt and isochronous transfers offer service guarantees because their bandwidth is reserved. Such "periodic" transfers are awkward to use through usbfs, unless you're using the asynchronous calls. However, interrupt transfers can also be used in a synchronous "one shot" style.

Your user-mode driver should never need to worry about cleaning up request state when the device is disconnected, although it should close its open file descriptors as soon as it starts seeing the ENODEV errors.

The ioctl() Requests

To use these ioctls, you need to include the following headers in your userspace program:

```
#include <linux/usb.h>
#include <linux/usbdevice_fs.h>
#include <asm/byteorder.h>
```

The standard USB device model requests, from "Chapter 9" of the USB 2.0 specification, are automatically included from the `<linux/usb/ch9.h>` header.

Unless noted otherwise, the ioctl requests described here will update the modification time on the usbfs file to which they are applied (unless they fail). A return of zero indicates success; otherwise, a standard USB error code is returned (These are documented in USB Error codes).

Each of these files multiplexes access to several I/O streams, one per endpoint. Each device has one control endpoint (endpoint zero) which supports a limited RPC style RPC access. Devices are configured by hub_wq (in the kernel) setting a device-wide configuration that affects things like power consumption and basic functionality. The endpoints are part of USB interfaces, which may have altsettings affecting things like which endpoints are available. Many devices only have a single configuration and interface, so drivers for them will ignore configurations and altsettings.

Management/Status Requests

A number of usbfs requests don't deal very directly with device I/O. They mostly relate to device management and status. These are all synchronous requests.

USBDEVFS_CLAIMINTERFACE This is used to force usbfs to claim a specific interface, which has not previously been claimed by usbfs or any other kernel driver. The ioctl parameter is an integer holding the number of the interface (bInterfaceNumber from descriptor).

Note that if your driver doesn't claim an interface before trying to use one of its endpoints, and no other driver has bound to it, then the interface is automatically claimed by usbfs.

This claim will be released by a RELEASEINTERFACE ioctl, or by closing the file descriptor. File modification time is not updated by this request.

USBDEVFS_CONNECTINFO Says whether the device is lowspeed. The ioctl parameter points to a structure like this:

```
struct usbdevfs_connectinfo {
        unsigned int   devnum;
        unsigned char  slow;
};
```

File modification time is not updated by this request.

You can't tell whether a "not slow" device is connected at high speed (480 MBit/sec) or just full speed (12 MBit/sec). You should know the devnum value already, it's the DDD value of the device file name.

USBDEVFS_GETDRIVER Returns the name of the kernel driver bound to a given interface (a string). Parameter is a pointer to this structure, which is modified:

```
struct usbdevfs_getdriver {
        unsigned int  interface;
        char          driver[USBDEVFS_MAXDRIVERNAME + 1];
};
```

File modification time is not updated by this request.

USBDEVFS_IOCTL Passes a request from userspace through to a kernel driver that has an ioctl entry in the struct usb_driver it registered:

```
struct usbdevfs_ioctl {
        int     ifno;
        int     ioctl_code;
        void    *data;
};

/* user mode call looks like this.
 * 'request' becomes the driver->ioctl() 'code' parameter.
 * the size of 'param' is encoded in 'request', and that data
 * is copied to or from the driver->ioctl() 'buf' parameter.
 */
static int
```

(continues on next page)

(continued from previous page)

```
usbdev_ioctl (int fd, int ifno, unsigned request, void *param)
{
        struct usbdevfs_ioctl   wrapper;

        wrapper.ifno = ifno;
        wrapper.ioctl_code = request;
        wrapper.data = param;

        return ioctl (fd, USBDEVFS_IOCTL, &wrapper);
}
```

File modification time is not updated by this request.

This request lets kernel drivers talk to user mode code through filesystem operations even when they don't create a character or block special device. It's also been used to do things like ask devices what device special file should be used. Two pre-defined ioctls are used to disconnect and reconnect kernel drivers, so that user mode code can completely manage binding and configuration of devices.

USBDEVFS_RELEASEINTERFACE This is used to release the claim usbfs made on interface, either implicitly or because of a USBDEVFS_CLAIMINTERFACE call, before the file descriptor is closed. The ioctl parameter is an integer holding the number of the interface (bInterfaceNumber from descriptor); File modification time is not updated by this request.

> **Warning:** No security check is made to ensure that the task which made the claim is the one which is releasing it. This means that user mode driver may interfere other ones.

USBDEVFS_RESETEP Resets the data toggle value for an endpoint (bulk or interrupt) to DATA0. The ioctl parameter is an integer endpoint number (1 to 15, as identified in the endpoint descriptor), with USB_DIR_IN added if the device's endpoint sends data to the host.

> **Warning:** Avoid using this request. It should probably be removed. Using it typically means the device and driver will lose toggle synchronization. If you really lost synchronization, you likely need to completely handshake with the device, using a request like CLEAR_HALT or SET_INTERFACE.

USBDEVFS_DROP_PRIVILEGES This is used to relinquish the ability to do certain operations which are considered to be privileged on a usbfs file descriptor. This includes claiming arbitrary interfaces, resetting a device on which there are currently claimed interfaces from other users, and issuing USBDEVFS_IOCTL calls. The ioctl parameter is a 32 bit mask of interfaces the user is allowed to claim on this file descriptor. You may issue this ioctl more than one time to narrow said mask.

Synchronous I/O Support

Synchronous requests involve the kernel blocking until the user mode request completes, either by finishing successfully or by reporting an error. In most cases this is the simplest way to use usbfs, although as noted above it does prevent performing I/O to more than one endpoint at a time.

USBDEVFS_BULK Issues a bulk read or write request to the device. The ioctl parameter is a pointer to this structure:

```
struct usbdevfs_bulktransfer {
        unsigned int  ep;
        unsigned int  len;
        unsigned int  timeout; /* in milliseconds */
        void          *data;
};
```

The `ep` value identifies a bulk endpoint number (1 to 15, as identified in an endpoint descriptor), masked with USB_DIR_IN when referring to an endpoint which sends data to the host from the device. The length of the data buffer is identified by `len`; Recent kernels support requests up to about 128KBytes. FIXME say how read length is returned, and how short reads are handled..

USBDEVFS_CLEAR_HALT Clears endpoint halt (stall) and resets the endpoint toggle. This is only meaningful for bulk or interrupt endpoints. The ioctl parameter is an integer endpoint number (1 to 15, as identified in an endpoint descriptor), masked with USB_DIR_IN when referring to an endpoint which sends data to the host from the device.

Use this on bulk or interrupt endpoints which have stalled, returning `-EPIPE` status to a data transfer request. Do not issue the control request directly, since that could invalidate the host's record of the data toggle.

USBDEVFS_CONTROL Issues a control request to the device. The ioctl parameter points to a structure like this:

```
struct usbdevfs_ctrltransfer {
        __u8   bRequestType;
        __u8   bRequest;
        __u16  wValue;
        __u16  wIndex;
        __u16  wLength;
        __u32  timeout;  /* in milliseconds */
        void   *data;
};
```

The first eight bytes of this structure are the contents of the SETUP packet to be sent to the device; see the USB 2.0 specification for details. The bRequestType value is composed by combining a `USB_TYPE_*` value, a `USB_DIR_*` value, and a `USB_RECIP_*` value (from `linux/usb.h`). If wLength is nonzero, it describes the length of the data buffer, which is either written to the device (USB_DIR_OUT) or read from the device (USB_DIR_IN).

At this writing, you can't transfer more than 4 KBytes of data to or from a device; usbfs has a limit, and some host controller drivers have a limit. (That'

s not usually a problem.) Also there's no way to say it's not OK to get a short read back from the device.

USBDEVFS_RESET Does a USB level device reset. The ioctl parameter is ignored. After the reset, this rebinds all device interfaces. File modification time is not updated by this request.

Warning: Avoid using this call until some usbcore bugs get fixed, since it does not fully synchronize device, interface, and driver (not just usbfs) state.

USBDEVFS_SETINTERFACE Sets the alternate setting for an interface. The ioctl parameter is a pointer to a structure like this:

```
struct usbdevfs_setinterface {
        unsigned int  interface;
        unsigned int  altsetting;
};
```

File modification time is not updated by this request.

Those struct members are from some interface descriptor applying to the current configuration. The interface number is the bInterfaceNumber value, and the altsetting number is the bAlternateSetting value. (This resets each endpoint in the interface.)

USBDEVFS_SETCONFIGURATION Issues the `usb_set_configuration()` call for the device. The parameter is an integer holding the number of a configuration (bConfigurationValue from descriptor). File modification time is not updated by this request.

Warning: Avoid using this call until some usbcore bugs get fixed, since it does not fully synchronize device, interface, and driver (not just usbfs) state.

Asynchronous I/O Support

As mentioned above, there are situations where it may be important to initiate concurrent operations from user mode code. This is particularly important for periodic transfers (interrupt and isochronous), but it can be used for other kinds of USB requests too. In such cases, the asynchronous requests described here are essential. Rather than submitting one request and having the kernel block until it completes, the blocking is separate.

These requests are packaged into a structure that resembles the URB used by kernel device drivers. (No POSIX Async I/O support here, sorry.) It identifies the endpoint type (`USBDEVFS_URB_TYPE_*`), endpoint (number, masked with USB_DIR_IN as appropriate), buffer and length, and a user "context" value serving to uniquely identify each request. (It's usually a pointer to per-request data.) Flags can modify requests (not as many as supported for kernel drivers).

Each request can specify a realtime signal number (between SIGRTMIN and SIGRTMAX, inclusive) to request a signal be sent when the request completes.

When usbfs returns these urbs, the status value is updated, and the buffer may have been modified. Except for isochronous transfers, the actual_length is updated to say how many bytes were transferred; if the USBDEVFS_URB_DISABLE_SPD flag is set (“short packets are not OK”), if fewer bytes were read than were requested then you get an error report:

```
struct usbdevfs_iso_packet_desc {
        unsigned int                    length;
        unsigned int                    actual_length;
        unsigned int                    status;
};

struct usbdevfs_urb {
        unsigned char                   type;
        unsigned char                   endpoint;
        int                             status;
        unsigned int                    flags;
        void                            *buffer;
        int                             buffer_length;
        int                             actual_length;
        int                             start_frame;
        int                             number_of_packets;
        int                             error_count;
        unsigned int                    signr;
        void                            *usercontext;
        struct usbdevfs_iso_packet_desc  iso_frame_desc[];
};
```

For these asynchronous requests, the file modification time reflects when the request was initiated. This contrasts with their use with the synchronous requests, where it reflects when requests complete.

USBDEVFS_DISCARDURB TBS File modification time is not updated by this request.

USBDEVFS_DISCSIGNAL TBS File modification time is not updated by this request.

USBDEVFS_REAPURB TBS File modification time is not updated by this request.

USBDEVFS_REAPURBNDELAY TBS File modification time is not updated by this request.

USBDEVFS_SUBMITURB TBS

20.1.8 The USB devices

The USB devices are now exported via debugfs:

- `/sys/kernel/debug/usb/devices` ⋯a text file showing each of the USB devices on known to the kernel, and their configuration descriptors. You can also poll() this to learn about new devices.

/sys/kernel/debug/usb/devices

This file is handy for status viewing tools in user mode, which can scan the text format and ignore most of it. More detailed device status (including class and vendor status) is available from device-specific files. For information about the current format of this file, see below.

This file, in combination with the poll() system call, can also be used to detect when devices are added or removed:

```
int fd;
struct pollfd pfd;

fd = open("/sys/kernel/debug/usb/devices", O_RDONLY);
pfd = { fd, POLLIN, 0 };
for (;;) {
    /* The first time through, this call will return immediately. */
    poll(&pfd, 1, -1);

    /* To see what's changed, compare the file's previous and current
       contents or scan the filesystem.  (Scanning is more precise.) */
}
```

Note that this behavior is intended to be used for informational and debug purposes. It would be more appropriate to use programs such as udev or HAL to initialize a device or start a user-mode helper program, for instance.

In this file, each device's output has multiple lines of ASCII output.

I made it ASCII instead of binary on purpose, so that someone can obtain some useful data from it without the use of an auxiliary program. However, with an auxiliary program, the numbers in the first 4 columns of each `T:` line (topology info: Lev, Prnt, Port, Cnt) can be used to build a USB topology diagram.

Each line is tagged with a one-character ID for that line:

```
T = Topology (etc.)
B = Bandwidth (applies only to USB host controllers, which are
virtualized as root hubs)
D = Device descriptor info.
P = Product ID info. (from Device descriptor, but they won't fit
together on one line)
S = String descriptors.
C = Configuration descriptor info. (* = active configuration)
I = Interface descriptor info.
E = Endpoint descriptor info.
```

/sys/kernel/debug/usb/devices output format

Legend:: d = decimal number (may have leading spaces or 0’s) x = hexadecimal number (may have leading spaces or 0’s) s = string

Topology info

```
T:  Bus=dd Lev=dd Prnt=dd Port=dd Cnt=dd Dev#=ddd Spd=dddd MxCh=dd
|   |      |      |       |       |      |         |        |__MaxChildren
|   |      |      |       |       |      |         |__Device Speed in Mbps
|   |      |      |       |       |      |__DeviceNumber
|   |      |      |       |       |__Count of devices at this level
|   |      |      |       |__Connector/Port on Parent for this device
|   |      |      |__Parent DeviceNumber
|   |      |__Level in topology for this bus
|   |__Bus number
|__Topology info tag
```

Speed may be:

1.5	Mbit/s for low speed USB
12	Mbit/s for full speed USB
480	Mbit/s for high speed USB (added for USB 2.0); also used for Wireless USB, which has no fixed speed
5000	Mbit/s for SuperSpeed USB (added for USB 3.0)

For reasons lost in the mists of time, the Port number is always too low by 1. For example, a device plugged into port 4 will show up with `Port=03`.

Bandwidth info

```
B:  Alloc=ddd/ddd us (xx%), #Int=ddd, #Iso=ddd
|   |                       |         |__Number of isochronous requests
|   |                       |__Number of interrupt requests
|   |__Total Bandwidth allocated to this bus
|__Bandwidth info tag
```

Bandwidth allocation is an approximation of how much of one frame (millisecond) is in use. It reflects only periodic transfers, which are the only transfers that reserve bandwidth. Control and bulk transfers use all other bandwidth, including reserved bandwidth that is not used for transfers (such as for short packets).

The percentage is how much of the “reserved” bandwidth is scheduled by those transfers. For a low or full speed bus (loosely, “USB 1.1”), 90% of the bus bandwidth is reserved. For a high speed bus (loosely, “USB 2.0”) 80% is reserved.

Device descriptor info & Product ID info

```
D:  Ver=x.xx Cls=xx(s) Sub=xx Prot=xx MxPS=dd #Cfgs=dd
P:  Vendor=xxxx ProdID=xxxx Rev=xx.xx
```

where:

```
D:  Ver=x.xx Cls=xx(sssss) Sub=xx Prot=xx MxPS=dd #Cfgs=dd
|   |        |             |      |       |       |__NumberConfigurations
|   |        |             |      |       |__MaxPacketSize of Default␣
↪Endpoint
|   |        |             |      |__DeviceProtocol
|   |        |             |__DeviceSubClass
|   |        |__DeviceClass
|   |__Device USB version
|__Device info tag #1
```

where:

```
P:  Vendor=xxxx ProdID=xxxx Rev=xx.xx
|   |           |           |__Product revision number
|   |           |__Product ID code
|   |__Vendor ID code
|__Device info tag #2
```

String descriptor info

```
S:  Manufacturer=ssss
|   |__Manufacturer of this device as read from the device.
|      For USB host controller drivers (virtual root hubs) this may
|      be omitted, or (for newer drivers) will identify the kernel
|      version and the driver which provides this hub emulation.
|__String info tag

S:  Product=ssss
|   |__Product description of this device as read from the device.
|      For older USB host controller drivers (virtual root hubs) this
|      indicates the driver; for newer ones, it's a product (and vendor)
|      description that often comes from the kernel's PCI ID database.
|__String info tag

S:  SerialNumber=ssss
|   |__Serial Number of this device as read from the device.
|      For USB host controller drivers (virtual root hubs) this is
|      some unique ID, normally a bus ID (address or slot name) that
|      can't be shared with any other device.
|__String info tag
```

Configuration descriptor info

```
C:* #Ifs=dd Cfg#=dd Atr=xx MPwr=dddmA
| | |       |       |      |__MaxPower in mA
| | |       |       |__Attributes
| | |       |__ConfigurationNumber
| | |__NumberOfInterfaces
| |__ "*" indicates the active configuration (others are " ")
|__Config info tag
```

USB devices may have multiple configurations, each of which act rather differently. For example, a bus-powered configuration might be much less capable than one that is self-powered. Only one device configuration can be active at a time; most devices have only one configuration.

Each configuration consists of one or more interfaces. Each interface serves a distinct "function", which is typically bound to a different USB device driver. One common example is a USB speaker with an audio interface for playback, and a HID interface for use with software volume control.

Interface descriptor info (can be multiple per Config)

```
I:* If#=dd Alt=dd #EPs=dd Cls=xx(sssss) Sub=xx Prot=xx Driver=ssss
| | |      |      |       |             |      |       |__Driver name
| | |      |      |       |             |      |          or "(none)"
| | |      |      |       |             |      |__InterfaceProtocol
| | |      |      |       |             |__InterfaceSubClass
| | |      |      |       |__InterfaceClass
| | |      |      |__NumberOfEndpoints
| | |      |__AlternateSettingNumber
| | |__InterfaceNumber
| |__ "*" indicates the active altsetting (others are " ")
|__Interface info tag
```

A given interface may have one or more "alternate" settings. For example, default settings may not use more than a small amount of periodic bandwidth. To use significant fractions of bus bandwidth, drivers must select a non-default altsetting.

Only one setting for an interface may be active at a time, and only one driver may bind to an interface at a time. Most devices have only one alternate setting per interface.

Endpoint descriptor info (can be multiple per Interface)

```
E:  Ad=xx(s) Atr=xx(ssss) MxPS=dddd Ivl=dddss
|   |        |            |         |__Interval (max) between transfers
|   |        |            |__EndpointMaxPacketSize
|   |        |__Attributes(EndpointType)
|   |__EndpointAddress(I=In,O=Out)
|__Endpoint info tag
```

The interval is nonzero for all periodic (interrupt or isochronous) endpoints. For high speed endpoints the transfer interval may be measured in microseconds rather than milliseconds.

For high speed periodic endpoints, the `EndpointMaxPacketSize` reflects the per-microframe data transfer size. For "high bandwidth" endpoints, that can reflect two or three packets (for up to 3KBytes every 125 usec) per endpoint.

With the Linux-USB stack, periodic bandwidth reservations use the transfer intervals and sizes provided by URBs, which can be less than those found in endpoint descriptor.

Usage examples

If a user or script is interested only in Topology info, for example, use something like `grep ^T: /sys/kernel/debug/usb/devices` for only the Topology lines. A command like `grep -i ^[tdp]: /sys/kernel/debug/usb/devices` can be used to list only the lines that begin with the characters in square brackets, where the valid characters are TDPCIE. With a slightly more able script, it can display any selected lines (for example, only T, D, and P lines) and change their output format. (The `procusb` Perl script is the beginning of this idea. It will list only selected lines [selected from TBDPSCIE] or "All" lines from `/sys/kernel/debug/usb/devices`.)

The Topology lines can be used to generate a graphic/pictorial of the USB devices on a system's root hub. (See more below on how to do this.)

The Interface lines can be used to determine what driver is being used for each device, and which altsetting it activated.

The Configuration lines could be used to list maximum power (in milliamps) that a system's USB devices are using. For example, `grep ^C: /sys/kernel/debug/usb/devices`.

Here's an example, from a system which has a UHCI root hub, an external hub connected to the root hub, and a mouse and a serial converter connected to the external hub.

```
T:  Bus=00 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12   MxCh= 2
B:  Alloc= 28/900 us ( 3%), #Int=  2, #Iso=  0
D:  Ver= 1.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 0.00
S:  Product=USB UHCI Root Hub
S:  SerialNumber=dce0
C:* #Ifs= 1 Cfg#= 1 Atr=40 MxPwr=  0mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   8 Ivl=255ms

T:  Bus=00 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12   MxCh= 4
D:  Ver= 1.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0451 ProdID=1446 Rev= 1.00
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   1 Ivl=255ms

T:  Bus=00 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#=  3 Spd=1.5  MxCh= 0
```

(continues on next page)

(continued from previous page)

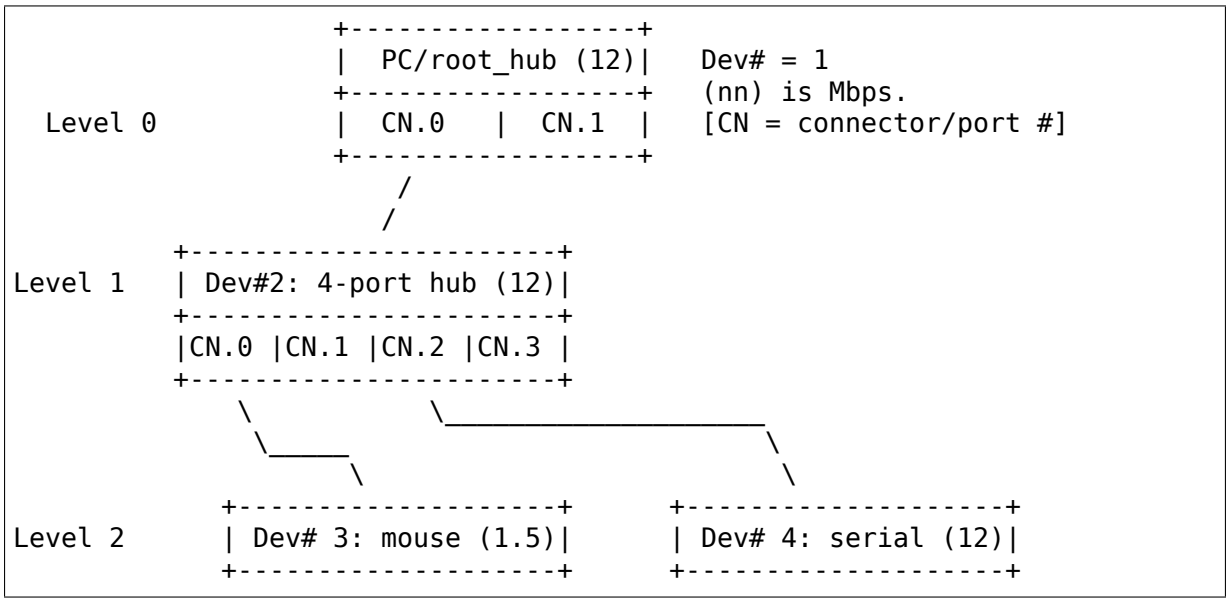
```
D:  Ver= 1.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=04b4 ProdID=0001 Rev= 0.00
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=01 Prot=02 Driver=mouse
E:  Ad=81(I) Atr=03(Int.) MxPS=   3 Ivl= 10ms

T:  Bus=00 Lev=02 Prnt=02 Port=02 Cnt=02 Dev#=  4 Spd=12   MxCh= 0
D:  Ver= 1.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0565 ProdID=0001 Rev= 1.08
S:  Manufacturer=Peracom Networks, Inc.
S:  Product=Peracom USB to Serial Converter
C:* #Ifs= 1 Cfg#= 1 Atr=a0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 3 Cls=00(>ifc ) Sub=00 Prot=00 Driver=serial
E:  Ad=81(I) Atr=02(Bulk) MxPS=  64 Ivl= 16ms
E:  Ad=01(O) Atr=02(Bulk) MxPS=  16 Ivl= 16ms
E:  Ad=82(I) Atr=03(Int.) MxPS=   8 Ivl=  8ms
```

Selecting only the `T:` and `I:` lines from this (for example, by using `procusb ti`), we have

```
T:  Bus=00 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12   MxCh= 2
T:  Bus=00 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12   MxCh= 4
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
T:  Bus=00 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#=  3 Spd=1.5  MxCh= 0
I:  If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=01 Prot=02 Driver=mouse
T:  Bus=00 Lev=02 Prnt=02 Port=02 Cnt=02 Dev#=  4 Spd=12   MxCh= 0
I:  If#= 0 Alt= 0 #EPs= 3 Cls=00(>ifc ) Sub=00 Prot=00 Driver=serial
```

Physically this looks like (or could be converted to):



Or, in a more tree-like structure (ports [Connectors] without connections could be omitted):

```
PC:  Dev# 1, root hub, 2 ports, 12 Mbps
|_ CN.0:  Dev# 2, hub, 4 ports, 12 Mbps
     |_ CN.0:  Dev #3, mouse, 1.5 Mbps
```

(continues on next page)

(continued from previous page)

```
     |_ CN.1:
     |_ CN.2:  Dev #4, serial, 12 Mbps
     |_ CN.3:
|_ CN.1:
```

20.2 USB Gadget API for Linux

Author David Brownell

Date 20 August 2004

20.2.1 Introduction

This document presents a Linux-USB "Gadget" kernel mode API, for use within peripherals and other USB devices that embed Linux. It provides an overview of the API structure, and shows how that fits into a system development project. This is the first such API released on Linux to address a number of important problems, including:

- Supports USB 2.0, for high speed devices which can stream data at several dozen megabytes per second.
- Handles devices with dozens of endpoints just as well as ones with just two fixed-function ones. Gadget drivers can be written so they're easy to port to new hardware.
- Flexible enough to expose more complex USB device capabilities such as multiple configurations, multiple interfaces, composite devices, and alternate interface settings.
- USB "On-The-Go" (OTG) support, in conjunction with updates to the Linux-USB host side.
- Sharing data structures and API models with the Linux-USB host side API. This helps the OTG support, and looks forward to more-symmetric frameworks (where the same I/O model is used by both host and device side drivers).
- Minimalist, so it's easier to support new device controller hardware. I/O processing doesn't imply large demands for memory or CPU resources.

Most Linux developers will not be able to use this API, since they have USB `host` hardware in a PC, workstation, or server. Linux users with embedded systems are more likely to have USB peripheral hardware. To distinguish drivers running inside such hardware from the more familiar Linux "USB device drivers", which are host side proxies for the real USB devices, a different term is used: the drivers inside the peripherals are "USB gadget drivers". In USB protocol interactions, the device driver is the master (or "client driver") and the gadget driver is the slave (or "function driver").

The gadget API resembles the host side Linux-USB API in that both use queues of request objects to package I/O buffers, and those requests may be submitted or

canceled. They share common definitions for the standard USB Chapter 9 messages, structures, and constants. Also, both APIs bind and unbind drivers to devices. The APIs differ in detail, since the host side's current URB framework exposes a number of implementation details and assumptions that are inappropriate for a gadget API. While the model for control transfers and configuration management is necessarily different (one side is a hardware-neutral master, the other is a hardware-aware slave), the endpoint I/0 API used here should also be usable for an overhead-reduced host side API.

20.2.2 Structure of Gadget Drivers

A system running inside a USB peripheral normally has at least three layers inside the kernel to handle USB protocol processing, and may have additional layers in user space code. The `gadget` API is used by the middle layer to interact with the lowest level (which directly handles hardware).

In Linux, from the bottom up, these layers are:

USB Controller Driver This is the lowest software level. It is the only layer that talks to hardware, through registers, fifos, dma, irqs, and the like. The `<linux/usb/gadget.h>` API abstracts the peripheral controller endpoint hardware. That hardware is exposed through endpoint objects, which accept streams of IN/OUT buffers, and through callbacks that interact with gadget drivers. Since normal USB devices only have one upstream port, they only have one of these drivers. The controller driver can support any number of different gadget drivers, but only one of them can be used at a time.

Examples of such controller hardware include the PCI-based NetChip 2280 USB 2.0 high speed controller, the SA-11x0 or PXA-25x UDC (found within many PDAs), and a variety of other products.

Gadget Driver The lower boundary of this driver implements hardware-neutral USB functions, using calls to the controller driver. Because such hardware varies widely in capabilities and restrictions, and is used in embedded environments where space is at a premium, the gadget driver is often configured at compile time to work with endpoints supported by one particular controller. Gadget drivers may be portable to several different controllers, using conditional compilation. (Recent kernels substantially simplify the work involved in supporting new hardware, by autoconfiguring endpoints automatically for many bulk-oriented drivers.) Gadget driver responsibilities include:

- handling setup requests (ep0 protocol responses) possibly including class-specific functionality
- returning configuration and string descriptors
- (re)setting configurations and interface altsettings, including enabling and configuring endpoints
- handling life cycle events, such as managing bindings to hardware, USB suspend/resume, remote wakeup, and disconnection from the USB host.
- managing IN and OUT transfers on all currently enabled endpoints

Such drivers may be modules of proprietary code, although that approach is discouraged in the Linux community.

Upper Level Most gadget drivers have an upper boundary that connects to some Linux driver or framework in Linux. Through that boundary flows the data which the gadget driver produces and/or consumes through protocol transfers over USB. Examples include:

- user mode code, using generic (gadgetfs) or application specific files in `/dev`
- networking subsystem (for network gadgets, like the CDC Ethernet Model gadget driver)
- data capture drivers, perhaps video4Linux or a scanner driver; or test and measurement hardware.
- input subsystem (for HID gadgets)
- sound subsystem (for audio gadgets)
- file system (for PTP gadgets)
- block i/o subsystem (for usb-storage gadgets)
- ···and more

Additional Layers Other layers may exist. These could include kernel layers, such as network protocol stacks, as well as user mode applications building on standard POSIX system call APIs such as `open()`, `close()`, `read()` and `write()`. On newer systems, POSIX Async I/O calls may be an option. Such user mode code will not necessarily be subject to the GNU General Public License (GPL).

OTG-capable systems will also need to include a standard Linux-USB host side stack, with `usbcore`, one or more Host Controller Drivers (HCDs), USB Device Drivers to support the OTG "Targeted Peripheral List", and so forth. There will also be an OTG Controller Driver, which is visible to gadget and device driver developers only indirectly. That helps the host and device side USB controllers implement the two new OTG protocols (HNP and SRP). Roles switch (host to peripheral, or vice versa) using HNP during USB suspend processing, and SRP can be viewed as a more battery-friendly kind of device wakeup protocol.

Over time, reusable utilities are evolving to help make some gadget driver tasks simpler. For example, building configuration descriptors from vectors of descriptors for the configurations interfaces and endpoints is now automated, and many drivers now use autoconfiguration to choose hardware endpoints and initialize their descriptors. A potential example of particular interest is code implementing standard USB-IF protocols for HID, networking, storage, or audio classes. Some developers are interested in KDB or KGDB hooks, to let target hardware be remotely debugged. Most such USB protocol code doesn't need to be hardware-specific, any more than network protocols like X11, HTTP, or NFS are. Such gadget-side interface drivers should eventually be combined, to implement composite devices.

20.2.3 Kernel Mode Gadget API

Gadget drivers declare themselves through a struct `usb_gadget_driver`, which is responsible for most parts of enumeration for a struct `usb_gadget`. The response to a set_configuration usually involves enabling one or more of the struct `usb_ep` objects exposed by the gadget, and submitting one or more struct `usb_request` buffers to transfer data. Understand those four data types, and their operations, and you will understand how this API works.

Note: Other than the "Chapter 9" data types, most of the significant data types and functions are described here.

However, some relevant information is likely omitted from what you are reading. One example of such information is endpoint autoconfiguration. You'll have to read the header file, and use example source code (such as that for "Gadget Zero"), to fully understand the API.

The part of the API implementing some basic driver capabilities is specific to the version of the Linux kernel that's in use. The 2.6 and upper kernel versions include a driver model framework that has no analogue on earlier kernels; so those parts of the gadget API are not fully portable. (They are implemented on 2.4 kernels, but in a different way.) The driver model state is another part of this API that is ignored by the kerneldoc tools.

The core API does not expose every possible hardware feature, only the most widely available ones. There are significant hardware features, such as device-to-device DMA (without temporary storage in a memory buffer) that would be added using hardware-specific APIs.

This API allows drivers to use conditional compilation to handle endpoint capabilities of different hardware, but doesn't require that. Hardware tends to have arbitrary restrictions, relating to transfer types, addressing, packet sizes, buffering, and availability. As a rule, such differences only matter for "endpoint zero" logic that handles device configuration and management. The API supports limited run-time detection of capabilities, through naming conventions for endpoints. Many drivers will be able to at least partially autoconfigure themselves. In particular, driver init sections will often have endpoint autoconfiguration logic that scans the hardware's list of endpoints to find ones matching the driver requirements (relying on those conventions), to eliminate some of the most common reasons for conditional compilation.

Like the Linux-USB host side API, this API exposes the "chunky" nature of USB messages: I/O requests are in terms of one or more "packets", and packet boundaries are visible to drivers. Compared to RS-232 serial protocols, USB resembles synchronous protocols like HDLC (N bytes per frame, multipoint addressing, host as the primary station and devices as secondary stations) more than asynchronous ones (tty style: 8 data bits per frame, no parity, one stop bit). So for example the controller drivers won't buffer two single byte writes into a single two-byte USB IN packet, although gadget drivers may do so when they implement protocols where packet boundaries (and "short packets") are not significant.

Driver Life Cycle

Gadget drivers make endpoint I/O requests to hardware without needing to know many details of the hardware, but driver setup/configuration code needs to handle some differences. Use the API like this:

1. Register a driver for the particular device side usb controller hardware, such as the net2280 on PCI (USB 2.0), sa11x0 or pxa25x as found in Linux PDAs, and so on. At this point the device is logically in the USB ch9 initial state (`attached`), drawing no power and not usable (since it does not yet support enumeration). Any host should not see the device, since it's not activated the data line pullup used by the host to detect a device, even if VBUS power is available.

2. Register a gadget driver that implements some higher level device function. That will then bind() to a `usb_gadget`, which activates the data line pullup sometime after detecting VBUS.

3. The hardware driver can now start enumerating. The steps it handles are to accept USB `power` and `set_address` requests. Other steps are handled by the gadget driver. If the gadget driver module is unloaded before the host starts to enumerate, steps before step 7 are skipped.

4. The gadget driver's `setup()` call returns usb descriptors, based both on what the bus interface hardware provides and on the functionality being implemented. That can involve alternate settings or configurations, unless the hardware prevents such operation. For OTG devices, each configuration descriptor includes an OTG descriptor.

5. The gadget driver handles the last step of enumeration, when the USB host issues a `set_configuration` call. It enables all endpoints used in that configuration, with all interfaces in their default settings. That involves using a list of the hardware's endpoints, enabling each endpoint according to its descriptor. It may also involve using `usb_gadget_vbus_draw` to let more power be drawn from VBUS, as allowed by that configuration. For OTG devices, setting a configuration may also involve reporting HNP capabilities through a user interface.

6. Do real work and perform data transfers, possibly involving changes to interface settings or switching to new configurations, until the device is disconnect()ed from the host. Queue any number of transfer requests to each endpoint. It may be suspended and resumed several times before being disconnected. On disconnect, the drivers go back to step 3 (above).

7. When the gadget driver module is being unloaded, the driver unbind() callback is issued. That lets the controller driver be unloaded.

Drivers will normally be arranged so that just loading the gadget driver module (or statically linking it into a Linux kernel) allows the peripheral device to be enumerated, but some drivers will defer enumeration until some higher level component (like a user mode daemon) enables it. Note that at this lowest level there are no policies about how ep0 configuration logic is implemented, except that it should obey USB specifications. Such issues are in the domain of gadget drivers, including knowing about implementation constraints imposed by some USB controllers or understanding that composite devices might happen to be built by integrating

reusable components.

Note that the lifecycle above can be slightly different for OTG devices. Other than providing an additional OTG descriptor in each configuration, only the HNP-related differences are particularly visible to driver code. They involve reporting requirements during the `SET_CONFIGURATION` request, and the option to invoke HNP during some suspend callbacks. Also, SRP changes the semantics of `usb_gadget_wakeup` slightly.

USB 2.0 Chapter 9 Types and Constants

Gadget drivers rely on common USB structures and constants defined in the linux/usb/ch9.h header file, which is standard in Linux 2.6+ kernels. These are the same types and constants used by host side drivers (and usbcore).

Core Objects and Methods

These are declared in `<linux/usb/gadget.h>`, and are used by gadget drivers to interact with USB peripheral controller drivers.

struct **usb_request**
describes one i/o request

Definition

```
struct usb_request {
  void *buf;
  unsigned length;
  dma_addr_t dma;
  struct scatterlist      *sg;
  unsigned num_sgs;
  unsigned num_mapped_sgs;
  unsigned stream_id:16;
  unsigned is_last:1;
  unsigned no_interrupt:1;
  unsigned zero:1;
  unsigned short_not_ok:1;
  unsigned dma_mapped:1;
  void (*complete)(struct usb_ep *ep, struct usb_request *req);
  void *context;
  struct list_head        list;
  unsigned frame_number;
  int status;
  unsigned actual;
};
```

Members

buf Buffer used for data. Always provide this; some controllers only use PIO, or don't use DMA for some endpoints.

length Length of that data

dma DMA address corresponding to 'buf'. If you don't set this field, and the usb controller needs one, it is responsible for mapping and unmapping the buffer.

sg a scatterlist for SG-capable controllers.

num_sgs number of SG entries

num_mapped_sgs number of SG entries mapped to DMA (internal)

stream_id The stream id, when USB3.0 bulk streams are being used

is_last Indicates if this is the last request of a stream_id before switching to a different stream (required for DWC3 controllers).

no_interrupt If true, hints that no completion irq is needed. Helpful sometimes with deep request queues that are handled directly by DMA controllers.

zero If true, when writing data, makes the last packet be "short" by adding a zero length packet as needed;

short_not_ok When reading data, makes short packets be treated as errors (queue stops advancing till cleanup).

dma_mapped Indicates if request has been mapped to DMA (internal)

complete Function called when request completes, so this request and its buffer may be re-used. The function will always be called with interrupts disabled, and it must not sleep. Reads terminate with a short packet, or when the buffer fills, whichever comes first. When writes terminate, some data bytes will usually still be in flight (often in a hardware fifo). Errors (for reads or writes) stop the queue from advancing until the completion function returns, so that any transfers invalidated by the error may first be dequeued.

context For use by the completion callback

list For use by the gadget driver.

frame_number Reports the interval number in (micro)frame in which the isochronous transfer was transmitted or received.

status Reports completion code, zero or a negative errno. Normally, faults block the transfer queue from advancing until the completion callback returns. Code "-ESHUTDOWN" indicates completion caused by device disconnect, or when the driver disabled the endpoint.

actual Reports bytes transferred to/from the buffer. For reads (OUT transfers) this may be less than the requested length. If the short_not_ok flag is set, short reads are treated as errors even when status otherwise indicates successful completion. Note that for writes (IN transfers) some data bytes may still reside in a device-side FIFO when the request is reported as complete.

Description

These are allocated/freed through the endpoint they're used with. The hardware's driver can add extra per-request data to the memory it returns, which often avoids separate memory allocations (potential failures), later when the request is queued.

Request flags affect request handling, such as whether a zero length packet is written (the "zero" flag), whether a short read should be treated as an error (blocking request queue advance, the "short_not_ok" flag), or hinting that an interrupt is not required (the "no_interrupt" flag, for use with deep request queues).

Bulk endpoints can use any size buffers, and can also be used for interrupt transfers. interrupt-only endpoints can be much less functional.

NOTE

this is analogous to ‘struct urb’ on the host side, except that it’s thinner and promotes more pre-allocation.

struct **usb_ep_caps**
endpoint capabilities description

Definition

```
struct usb_ep_caps {
  unsigned type_control:1;
  unsigned type_iso:1;
  unsigned type_bulk:1;
  unsigned type_int:1;
  unsigned dir_in:1;
  unsigned dir_out:1;
};
```

Members

type_control Endpoint supports control type (reserved for ep0).

type_iso Endpoint supports isochronous transfers.

type_bulk Endpoint supports bulk transfers.

type_int Endpoint supports interrupt transfers.

dir_in Endpoint supports IN direction.

dir_out Endpoint supports OUT direction.

struct **usb_ep**
device side representation of USB endpoint

Definition

```
struct usb_ep {
  void *driver_data;
  const char               *name;
  const struct usb_ep_ops *ops;
  struct list_head         ep_list;
  struct usb_ep_caps       caps;
  bool claimed;
  bool enabled;
  unsigned maxpacket:16;
  unsigned maxpacket_limit:16;
  unsigned max_streams:16;
  unsigned mult:2;
  unsigned maxburst:5;
  u8 address;
  const struct usb_endpoint_descriptor    *desc;
  const struct usb_ss_ep_comp_descriptor  *comp_desc;
};
```

Members

driver_data for use by the gadget driver.

name identifier for the endpoint, such as “ep-a” or “ep9in-bulk”

ops Function pointers used to access hardware-specific operations.

ep_list the gadget’s ep_list holds all of its endpoints

caps The structure describing types and directions supported by endoint.

claimed True if this endpoint is claimed by a function.

enabled The current endpoint enabled/disabled state.

maxpacket The maximum packet size used on this endpoint. The initial value can sometimes be reduced (hardware allowing), according to the endpoint descriptor used to configure the endpoint.

maxpacket_limit The maximum packet size value which can be handled by this endpoint. It’s set once by UDC driver when endpoint is initialized, and should not be changed. Should not be confused with maxpacket.

max_streams The maximum number of streams supported by this EP (0 - 16, actual number is 2^n)

mult multiplier, ‘mult’ value for SS Isoc EPs

maxburst the maximum number of bursts supported by this EP (for usb3)

address used to identify the endpoint when finding descriptor that matches connection speed

desc endpoint descriptor. This pointer is set before the endpoint is enabled and remains valid until the endpoint is disabled.

comp_desc In case of SuperSpeed support, this is the endpoint companion descriptor that is used to configure the endpoint

Description

the bus controller driver lists all the general purpose endpoints in gadget->ep_list. the control endpoint (gadget->ep0) is not in that list, and is accessed only in response to a driver setup() callback.

struct **usb_gadget**
represents a usb slave device

Definition

```
struct usb_gadget {
  struct work_struct              work;
  struct usb_udc                  *udc;
  const struct usb_gadget_ops     *ops;
  struct usb_ep                   *ep0;
  struct list_head                ep_list;
  enum usb_device_speed           speed;
  enum usb_device_speed           max_speed;
  enum usb_device_state           state;
  const char                      *name;
  struct device                   dev;
  unsigned isoch_delay;
```

(continues on next page)

(continued from previous page)

```
  unsigned out_epnum;
  unsigned in_epnum;
  unsigned mA;
  struct usb_otg_caps             *otg_caps;
  unsigned sg_supported:1;
  unsigned is_otg:1;
  unsigned is_a_peripheral:1;
  unsigned b_hnp_enable:1;
  unsigned a_hnp_support:1;
  unsigned a_alt_hnp_support:1;
  unsigned hnp_polling_support:1;
  unsigned host_request_flag:1;
  unsigned quirk_ep_out_aligned_size:1;
  unsigned quirk_altset_not_supp:1;
  unsigned quirk_stall_not_supp:1;
  unsigned quirk_zlp_not_supp:1;
  unsigned quirk_avoids_skb_reserve:1;
  unsigned is_selfpowered:1;
  unsigned deactivated:1;
  unsigned connected:1;
  unsigned lpm_capable:1;
  int irq;
};
```

Members

work (internal use) Workqueue to be used for sysfs_notify()

udc struct usb_udc pointer for this gadget

ops Function pointers used to access hardware-specific operations.

ep0 Endpoint zero, used when reading or writing responses to driver setup() requests

ep_list List of other endpoints supported by the device.

speed Speed of current connection to USB host.

max_speed Maximal speed the UDC can handle. UDC must support this and all slower speeds.

state the state we are now (attached, suspended, configured, etc)

name Identifies the controller hardware type. Used in diagnostics and sometimes configuration.

dev Driver model state for this abstract device.

isoch_delay value from Set Isoch Delay request. Only valid on SS/SSP

out_epnum last used out ep number

in_epnum last used in ep number

mA last set mA value

otg_caps OTG capabilities of this gadget.

sg_supported true if we can handle scatter-gather

is_otg True if the USB device port uses a Mini-AB jack, so that the gadget driver must provide a USB OTG descriptor.

is_a_peripheral False unless is_otg, the "A" end of a USB cable is in the Mini-AB jack, and HNP has been used to switch roles so that the "A" device currently acts as A-Peripheral, not A-Host.

b_hnp_enable OTG device feature flag, indicating that the A-Host enabled HNP support.

a_hnp_support OTG device feature flag, indicating that the A-Host supports HNP at this port.

a_alt_hnp_support OTG device feature flag, indicating that the A-Host only supports HNP on a different root port.

hnp_polling_support OTG device feature flag, indicating if the OTG device in peripheral mode can support HNP polling.

host_request_flag OTG device feature flag, indicating if A-Peripheral or B-Peripheral wants to take host role.

quirk_ep_out_aligned_size epout requires buffer size to be aligned to MaxPacketSize.

quirk_altset_not_supp UDC controller doesn't support alt settings.

quirk_stall_not_supp UDC controller doesn't support stalling.

quirk_zlp_not_supp UDC controller doesn't support ZLP.

quirk_avoids_skb_reserve udc/platform wants to avoid skb_reserve() in u_ether.c to improve performance.

is_selfpowered if the gadget is self-powered.

deactivated True if gadget is deactivated - in deactivated state it cannot be connected.

connected True if gadget is connected.

lpm_capable If the gadget max_speed is FULL or HIGH, this flag indicates that it supports LPM as per the LPM ECN & errata.

irq the interrupt number for device controller.

Description

Gadgets have a mostly-portable "gadget driver" implementing device functions, handling all usb configurations and interfaces. Gadget drivers talk to hardware-specific code indirectly, through ops vectors. That insulates the gadget driver from hardware details, and packages the hardware endpoints through generic i/o queues. The "usb_gadget" and "usb_ep" interfaces provide that insulation from the hardware.

Except for the driver data, all fields in this structure are read-only to the gadget driver. That driver data is part of the "driver model" infrastructure in 2.6 (and later) kernels, and for earlier systems is grouped in a similar structure that's not known to the rest of the kernel.

Values of the three OTG device feature flags are updated before the setup() call corresponding to USB_REQ_SET_CONFIGURATION, and before driver suspend() calls. They are valid only when is_otg, and when the device is acting as a B-Peripheral (so is_a_peripheral is false).

size_t **usb_ep_align**(struct usb_ep * ep, size_t len)
returns **len** aligned to ep's maxpacketsize.

Parameters

struct usb_ep * ep the endpoint whose maxpacketsize is used to align **len**

size_t len buffer size's length to align to **ep**'s maxpacketsize

Description

This helper is used to align buffer's size to an ep's maxpacketsize.

size_t **usb_ep_align_maybe**(struct usb_gadget * g, struct usb_ep * ep, size_t len)
returns **len** aligned to ep's maxpacketsize if gadget requires quirk_ep_out_aligned_size, otherwise returns len.

Parameters

struct usb_gadget * g controller to check for quirk

struct usb_ep * ep the endpoint whose maxpacketsize is used to align **len**

size_t len buffer size's length to align to **ep**'s maxpacketsize

Description

This helper is used in case it's required for any reason to check and maybe align buffer's size to an ep's maxpacketsize.

int **gadget_is_altset_supported**(struct usb_gadget * g)
return true iff the hardware supports altsettings

Parameters

struct usb_gadget * g controller to check for quirk

int **gadget_is_stall_supported**(struct usb_gadget * g)
return true iff the hardware supports stalling

Parameters

struct usb_gadget * g controller to check for quirk

int **gadget_is_zlp_supported**(struct usb_gadget * g)
return true iff the hardware supports zlp

Parameters

struct usb_gadget * g controller to check for quirk

int **gadget_avoids_skb_reserve**(struct usb_gadget * g)
return true iff the hardware would like to avoid skb_reserve to improve performance.

Parameters

struct usb_gadget * g controller to check for quirk

int **gadget_is_dualspeed**(struct usb_gadget * g)
return true iff the hardware handles high speed

Parameters

struct usb_gadget * g controller that might support both high and full speeds

int **gadget_is_superspeed**(struct usb_gadget * g)
return true if the hardware handles superspeed

Parameters

struct usb_gadget * g controller that might support superspeed

int **gadget_is_superspeed_plus**(struct usb_gadget * g)
return true if the hardware handles superspeed plus

Parameters

struct usb_gadget * g controller that might support superspeed plus

int **gadget_is_otg**(struct usb_gadget * g)
return true iff the hardware is OTG-ready

Parameters

struct usb_gadget * g controller that might have a Mini-AB connector

Description

This is a runtime test, since kernels with a USB-OTG stack sometimes run on boards which only have a Mini-B (or Mini-A) connector.

struct **usb_gadget_driver**
driver for usb 'slave' devices

Definition

```
struct usb_gadget_driver {
  char *function;
  enum usb_device_speed   max_speed;
  int (*bind)(struct usb_gadget *gadget, struct usb_gadget_driver *driver);
  void (*unbind)(struct usb_gadget *);
  int (*setup)(struct usb_gadget *, const struct usb_ctrlrequest *);
  void (*disconnect)(struct usb_gadget *);
  void (*suspend)(struct usb_gadget *);
  void (*resume)(struct usb_gadget *);
  void (*reset)(struct usb_gadget *);
  struct device_driver    driver;
  char *udc_name;
  struct list_head        pending;
  unsigned match_existing_only:1;
};
```

Members

function String describing the gadget's function

max_speed Highest speed the driver handles.

bind the driver's bind callback

unbind Invoked when the driver is unbound from a gadget, usually from rmmod (after a disconnect is reported). Called in a context that permits sleeping.

setup Invoked for ep0 control requests that aren't handled by the hardware level driver. Most calls must be handled by the gadget driver, including descriptor and configuration management. The 16 bit members of the setup data are in USB byte order. Called in_interrupt; this may not sleep. Driver queues a response to ep0, or returns negative to stall.

disconnect Invoked after all transfers have been stopped, when the host is disconnected. May be called in_interrupt; this may not sleep. Some devices can't detect disconnect, so this might not be called except as part of controller shutdown.

suspend Invoked on USB suspend. May be called in_interrupt.

resume Invoked on USB resume. May be called in_interrupt.

reset Invoked on USB bus reset. It is mandatory for all gadget drivers and should be called in_interrupt.

driver Driver model state for this driver.

udc_name A name of UDC this driver should be bound to. If udc_name is NULL, this driver will be bound to any available UDC.

pending UDC core private data used for deferred probe of this driver.

match_existing_only If udc is not found, return an error and don't add this gadget driver to list of pending driver

Description

Devices are disabled till a gadget driver successfully bind()s, which means the driver will handle setup() requests needed to enumerate (and meet "chapter 9" requirements) then do some useful work.

If gadget->is_otg is true, the gadget driver must provide an OTG descriptor during enumeration, or else fail the bind() call. In such cases, no USB traffic may flow until both bind() returns without having called usb_gadget_disconnect(), and the USB host stack has initialized.

Drivers use hardware-specific knowledge to configure the usb hardware. endpoint addressing is only one of several hardware characteristics that are in descriptors the ep0 implementation returns from setup() calls.

Except for ep0 implementation, most driver code shouldn't need change to run on top of different usb controllers. It'll use endpoints set up by that ep0 implementation.

The usb controller driver handles a few standard usb requests. Those include set_address, and feature flags for devices, interfaces, and endpoints (the get_status, set_feature, and clear_feature requests).

Accordingly, the driver's setup() callback must always implement all get_descriptor requests, returning at least a device descriptor and a configuration descriptor. Drivers must make sure the endpoint descriptors match any hardware constraints. Some hardware also constrains other descriptors. (The pxa250 allows only configurations 1, 2, or 3).

The driver's setup() callback must also implement set_configuration, and should also implement set_interface, get_configuration, and get_interface. Setting a configuration (or interface) is where endpoints should be activated or (config 0) shut down.

(Note that only the default control endpoint is supported. Neither hosts nor devices generally support control traffic except to ep0.)

Most devices will ignore USB suspend/resume operations, and so will not provide those callbacks. However, some may need to change modes when the host is not longer directing those activities. For example, local controls (buttons, dials, etc) may need to be re-enabled since the (remote) host can't do that any longer; or an error state might be cleared, to make the device behave identically whether or not power is maintained.

int **usb_gadget_probe_driver**(struct usb_gadget_driver * driver)
probe a gadget driver

Parameters

struct usb_gadget_driver * driver the driver being registered

Context

can sleep

Description

Call this in your gadget driver's module initialization function, to tell the underlying usb controller driver about your driver. The **bind()** function will be called to bind it to a gadget before this registration call returns. It's expected that the **bind()** function will be in init sections.

int **usb_gadget_unregister_driver**(struct usb_gadget_driver * driver)
unregister a gadget driver

Parameters

struct usb_gadget_driver * driver the driver being unregistered

Context

can sleep

Description

Call this in your gadget driver's module cleanup function, to tell the underlying usb controller that your driver is going away. If the controller is connected to a USB host, it will first disconnect(). The driver is also requested to unbind() and clean up any device state, before this procedure finally returns. It's expected that the unbind() functions will in in exit sections, so may not be linked in some kernels.

struct **usb_string**
wraps a C string and its USB id

Definition

```
struct usb_string {
  u8 id;
  const char              *s;
};
```

Members

id the (nonzero) ID for this string

s the string, in UTF-8 encoding

Description

If you're using `usb_gadget_get_string()`, use this to wrap a string together with its ID.

struct **usb_gadget_strings**
a set of USB strings in a given language

Definition

```
struct usb_gadget_strings {
  u16 language;
  struct usb_string       *strings;
};
```

Members

language identifies the strings' language (0x0409 for en-us)

strings array of strings with their ids

Description

If you're using `usb_gadget_get_string()`, use this to wrap all the strings for a given language.

void **usb_free_descriptors**(struct usb_descriptor_header ** v)
free descriptors returned by `usb_copy_descriptors()`

Parameters

struct usb_descriptor_header ** v vector of descriptors

Optional Utilities

The core API is sufficient for writing a USB Gadget Driver, but some optional utilities are provided to simplify common tasks. These utilities include endpoint autoconfiguration.

int **usb_gadget_get_string**(const struct usb_gadget_strings * table, int id, u8 * buf)
fill out a string descriptor

Parameters

const struct usb_gadget_strings * table of c strings encoded using UTF-8

int id string id, from low byte of wValue in get string descriptor

u8 * buf at least 256 bytes, must be 16-bit aligned

Description

Finds the UTF-8 string matching the ID, and converts it into a string descriptor in utf16-le. Returns length of descriptor (always even) or negative errno

If your driver needs stings in multiple languages, you'll probably "switch (wIndex) { ···}" in your ep0 string descriptor logic, using this routine after choosing which set of UTF-8 strings to use. Note that US-ASCII is a strict subset of UTF-8; any string bytes with the eighth bit set will be multibyte UTF-8 characters, not ISO-8859/1 characters (which are also widely used in C strings).

bool **usb_validate_langid**(u16 langid)
validate usb language identifiers

Parameters

u16 langid undescribed

Description

Returns true for valid language identifier, otherwise false.

int **usb_descriptor_fillbuf**(void * buf, unsigned buflen, const struct usb_descriptor_header ** src)
fill buffer with descriptors

Parameters

void * buf Buffer to be filled

unsigned buflen Size of buf

const struct usb_descriptor_header ** src Array of descriptor pointers, terminated by null pointer.

Description

Copies descriptors into the buffer, returning the length or a negative error code if they can't all be copied. Useful when assembling descriptors for an associated set of interfaces used as part of configuring a composite device; or in other cases where sets of descriptors need to be marshaled.

int **usb_gadget_config_buf**(const struct usb_config_descriptor * config, void * buf, unsigned length, const struct usb_descriptor_header ** desc)
builts a complete configuration descriptor

Parameters

const struct usb_config_descriptor * config Header for the descriptor, including characteristics such as power requirements and number of interfaces.

void * buf Buffer for the resulting configuration descriptor.

unsigned length Length of buffer. If this is not big enough to hold the entire configuration descriptor, an error code will be returned.

const struct usb_descriptor_header ** desc Null-terminated vector of pointers to the descriptors (interface, endpoint, etc) defining all functions in this device configuration.

Description

This copies descriptors into the response buffer, building a descriptor for that configuration. It returns the buffer length or a negative status code. The config.wTotalLength field is set to match the length of the result, but other descriptor fields (including power usage and interface count) must be set by the caller.

Gadget drivers could use this when constructing a config descriptor in response to USB_REQ_GET_DESCRIPTOR. They will need to patch the resulting bDescriptorType value if USB_DT_OTHER_SPEED_CONFIG is needed.

struct usb_descriptor_header ** **usb_copy_descriptors**(struct usb_descriptor_header ** src)

copy a vector of USB descriptors

Parameters

struct usb_descriptor_header ** src null-terminated vector to copy

Context

initialization code, which may sleep

Description

This makes a copy of a vector of USB descriptors. Its primary use is to support usb_function objects which can have multiple copies, each needing different descriptors. Functions may have static tables of descriptors, which are used as templates and customized with identifiers (for interfaces, strings, endpoints, and more) as needed by a given function instance.

Composite Device Framework

The core API is sufficient for writing drivers for composite USB devices (with more than one function in a given configuration), and also multi-configuration devices (also more than one function, but not necessarily sharing a given configuration). There is however an optional framework which makes it easier to reuse and combine functions.

Devices using this framework provide a struct `usb_composite_driver`, which in turn provides one or more struct `usb_configuration` instances. Each such configuration includes at least one struct `usb_function`, which packages a user visible role such as “network link” or “mass storage device”. Management functions may also exist, such as “Device Firmware Upgrade”.

struct **usb_os_desc_ext_prop**

describes one “Extended Property”

Definition

```
struct usb_os_desc_ext_prop {
  struct list_head        entry;
```

(continues on next page)

(continued from previous page)

```
  u8 type;
  int name_len;
  char *name;
  int data_len;
  char *data;
  struct config_item      item;
};
```

Members

entry used to keep a list of extended properties

type Extended Property type

name_len Extended Property unicode name length, including terminating '0'

name Extended Property name

data_len Length of Extended Property blob (for unicode store double len)

data Extended Property blob

item Represents this Extended Property in configfs

struct **usb_os_desc**
describes OS descriptors associated with one interface

Definition

```
struct usb_os_desc {
  char *ext_compat_id;
  struct list_head        ext_prop;
  int ext_prop_len;
  int ext_prop_count;
  struct mutex            *opts_mutex;
  struct config_group     group;
  struct module           *owner;
};
```

Members

ext_compat_id 16 bytes of "Compatible ID" and "Subcompatible ID"

ext_prop Extended Properties list

ext_prop_len Total length of Extended Properties blobs

ext_prop_count Number of Extended Properties

opts_mutex Optional mutex protecting config data of a usb_function_instance

group Represents OS descriptors associated with an interface in configfs

owner Module associated with this OS descriptor

struct **usb_os_desc_table**
describes OS descriptors associated with one interface of a usb_function

Definition

```
struct usb_os_desc_table {
  int if_id;
  struct usb_os_desc      *os_desc;
};
```

Members

if_id Interface id

os_desc “Extended Compatibility ID” and “Extended Properties” of the interface

Description

Each interface can have at most one “Extended Compatibility ID” and a number of “Extended Properties”.

struct **usb_function**
describes one function of a configuration

Definition

```
struct usb_function {
  const char                      *name;
  struct usb_gadget_strings       **strings;
  struct usb_descriptor_header    **fs_descriptors;
  struct usb_descriptor_header    **hs_descriptors;
  struct usb_descriptor_header    **ss_descriptors;
  struct usb_descriptor_header    **ssp_descriptors;
  struct usb_configuration        *config;
  struct usb_os_desc_table        *os_desc_table;
  unsigned os_desc_n;
  int (*bind)(struct usb_configuration *, struct usb_function *);
  void (*unbind)(struct usb_configuration *, struct usb_function *);
  void (*free_func)(struct usb_function *f);
  struct module           *mod;
  int (*set_alt)(struct usb_function *, unsigned interface, unsigned alt);
  int (*get_alt)(struct usb_function *, unsigned interface);
  void (*disable)(struct usb_function *);
  int (*setup)(struct usb_function *, const struct usb_ctrlrequest *);
  bool (*req_match)(struct usb_function *,const struct usb_ctrlrequest *,␣
↪bool config0);
  void (*suspend)(struct usb_function *);
  void (*resume)(struct usb_function *);
  int (*get_status)(struct usb_function *);
  int (*func_suspend)(struct usb_function *, u8 suspend_opt);
};
```

Members

name For diagnostics, identifies the function.

strings tables of strings, keyed by identifiers assigned during bind() and by language IDs provided in control requests

fs_descriptors Table of full (or low) speed descriptors, using interface and string identifiers assigned during **bind()**. If this pointer is null, the function will not be available at full speed (or at low speed).

hs_descriptors Table of high speed descriptors, using interface and string identifiers assigned during **bind()**. If this pointer is null, the function will not be available at high speed.

ss_descriptors Table of super speed descriptors, using interface and string identifiers assigned during **bind()**. If this pointer is null after initiation, the function will not be available at super speed.

ssp_descriptors Table of super speed plus descriptors, using interface and string identifiers assigned during **bind()**. If this pointer is null after initiation, the function will not be available at super speed plus.

config assigned when **usb_add_function()** is called; this is the configuration with which this function is associated.

os_desc_table Table of (interface id, os descriptors) pairs. The function can expose more than one interface. If an interface is a member of an IAD, only the first interface of IAD has its entry in the table.

os_desc_n Number of entries in os_desc_table

bind Before the gadget can register, all of its functions bind() to the available resources including string and interface identifiers used in interface or class descriptors; endpoints; I/O buffers; and so on.

unbind Reverses **bind**; called as a side effect of unregistering the driver which added this function.

free_func free the struct usb_function.

mod (internal) points to the module that created this structure.

set_alt (REQUIRED) Reconfigures altsettings; function drivers may initialize usb_ep.driver data at this time (when it is used). Note that setting an interface to its current altsetting resets interface state, and that all interfaces have a disabled state.

get_alt Returns the active altsetting. If this is not provided, then only altsetting zero is supported.

disable (REQUIRED) Indicates the function should be disabled. Reasons include host resetting or reconfiguring the gadget, and disconnection.

setup Used for interface-specific control requests.

req_match Tests if a given class request can be handled by this function.

suspend Notifies functions when the host stops sending USB traffic.

resume Notifies functions when the host restarts USB traffic.

get_status Returns function status as a reply to GetStatus() request when the recipient is Interface.

func_suspend callback to be called when SetFeature(FUNCTION_SUSPEND) is reseived

Description

A single USB function uses one or more interfaces, and should in most cases support operation at both full and high speeds. Each function is associated by

usb_add_function() with a one configuration; that function causes **bind()** to be called so resources can be allocated as part of setting up a gadget driver. Those resources include endpoints, which should be allocated using **usb_ep_autoconfig()**.

To support dual speed operation, a function driver provides descriptors for both high and full speed operation. Except in rare cases that don't involve bulk endpoints, each speed needs different endpoint descriptors.

Function drivers choose their own strategies for managing instance data. The simplest strategy just declares it "static", which means the function can only be activated once. If the function needs to be exposed in more than one configuration at a given speed, it needs to support multiple usb_function structures (one for each configuration).

A more complex strategy might encapsulate a **usb_function** structure inside a driver-specific instance structure to allows multiple activations. An example of multiple activations might be a CDC ACM function that supports two or more distinct instances within the same configuration, providing several independent logical data links to a USB host.

struct **usb_configuration**
represents one gadget configuration

Definition

```
struct usb_configuration {
  const char                    *label;
  struct usb_gadget_strings     **strings;
  const struct usb_descriptor_header **descriptors;
  void (*unbind)(struct usb_configuration *);
  int (*setup)(struct usb_configuration *, const struct usb_ctrlrequest *);
  u8 bConfigurationValue;
  u8 iConfiguration;
  u8 bmAttributes;
  u16 MaxPower;
  struct usb_composite_dev      *cdev;
};
```

Members

label For diagnostics, describes the configuration.

strings Tables of strings, keyed by identifiers assigned during **bind()** and by language IDs provided in control requests.

descriptors Table of descriptors preceding all function descriptors. Examples include OTG and vendor-specific descriptors.

unbind Reverses **bind**; called as a side effect of unregistering the driver which added this configuration.

setup Used to delegate control requests that aren't handled by standard device infrastructure or directed at a specific interface.

bConfigurationValue Copied into configuration descriptor.

iConfiguration Copied into configuration descriptor.

bmAttributes Copied into configuration descriptor.

MaxPower Power consumtion in mA. Used to compute bMaxPower in the configuration descriptor after considering the bus speed.

cdev assigned by **usb_add_config()** before calling **bind()**; this is the device associated with this configuration.

Description

Configurations are building blocks for gadget drivers structured around function drivers. Simple USB gadgets require only one function and one configuration, and handle dual-speed hardware by always providing the same functionality. Slightly more complex gadgets may have more than one single-function configuration at a given speed; or have configurations that only work at one speed.

Composite devices are, by definition, ones with configurations which include more than one function.

The lifecycle of a usb_configuration includes allocation, initialization of the fields described above, and calling **usb_add_config()** to set up internal data and bind it to a specific device. The configuration's **bind()** method is then used to initialize all the functions and then call **usb_add_function()** for them.

Those functions would normally be independent of each other, but that's not mandatory. CDC WMC devices are an example where functions often depend on other functions, with some functions subsidiary to others. Such interdependency may be managed in any way, so long as all of the descriptors complete by the time the composite driver returns from its bind() routine.

struct **usb_composite_driver**
groups configurations into a gadget

Definition

```
struct usb_composite_driver {
  const char                              *name;
  const struct usb_device_descriptor      *dev;
  struct usb_gadget_strings               **strings;
  enum usb_device_speed                   max_speed;
  unsigned needs_serial:1;
  int (*bind)(struct usb_composite_dev *cdev);
  int (*unbind)(struct usb_composite_dev *);
  void (*disconnect)(struct usb_composite_dev *);
  void (*suspend)(struct usb_composite_dev *);
  void (*resume)(struct usb_composite_dev *);
  struct usb_gadget_driver                gadget_driver;
};
```

Members

name For diagnostics, identifies the driver.

dev Template descriptor for the device, including default device identifiers.

strings tables of strings, keyed by identifiers assigned during **bind** and language IDs provided in control requests. Note: The first entries are predefined. The first entry that may be used is USB_GADGET_FIRST_AVAIL_IDX

max_speed Highest speed the driver supports.

needs_serial set to 1 if the gadget needs userspace to provide a serial number. If one is not provided, warning will be printed.

bind (REQUIRED) Used to allocate resources that are shared across the whole device, such as string IDs, and add its configurations using **usb_add_config()**. This may fail by returning a negative errno value; it should return zero on successful initialization.

unbind Reverses **bind**; called as a side effect of unregistering this driver.

disconnect optional driver disconnect method

suspend Notifies when the host stops sending USB traffic, after function notifications

resume Notifies configuration when the host restarts USB traffic, before function notifications

gadget_driver Gadget driver controlling this driver

Description

Devices default to reporting self powered operation. Devices which rely on bus powered operation should report this in their **bind** method.

Before returning from **bind**, various fields in the template descriptor may be overridden. These include the idVendor/idProduct/bcdDevice values normally to bind the appropriate host side driver, and the three strings (iManufacturer, iProduct, iSerialNumber) normally used to provide user meaningful device identifiers. (The strings will not be defined unless they are defined in **dev** and **strings**.) The correct ep0 maxpacket size is also reported, as defined by the underlying controller driver.

module_usb_composite_driver(__usb_composite_driver)
Helper macro for registering a USB gadget composite driver

Parameters

__usb_composite_driver usb_composite_driver struct

Description

Helper macro for USB gadget composite drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init()` and `module_exit()`

struct **usb_composite_dev**
represents one composite usb gadget

Definition

```
struct usb_composite_dev {
  struct usb_gadget               *gadget;
  struct usb_request              *req;
  struct usb_request              *os_desc_req;
  struct usb_configuration        *config;
  u8 qw_sign[OS_STRING_QW_SIGN_LEN];
  u8 b_vendor_code;
  struct usb_configuration        *os_desc_config;
  unsigned int                    use_os_string:1;
```

(continues on next page)

(continued from previous page)

```
  unsigned int                    setup_pending:1;
  unsigned int                    os_desc_pending:1;
};
```

Members

gadget read-only, abstracts the gadget's usb peripheral controller

req used for control responses; buffer is pre-allocated

os_desc_req used for OS descriptors responses; buffer is pre-allocated

config the currently active configuration

qw_sign qwSignature part of the OS string

b_vendor_code bMS_VendorCode part of the OS string

os_desc_config the configuration to be used with OS descriptors

use_os_string false by default, interested gadgets set it

setup_pending true when setup request is queued but not completed

os_desc_pending true when os_desc request is queued but not completed

Description

One of these devices is allocated and initialized before the associated device driver's bind() is called.

OPEN ISSUE: it appears that some WUSB devices will need to be built by combining a normal (wired) gadget with a wireless one. This revision of the gadget framework should probably try to make sure doing that won't hurt too much.

One notion for how to handle Wireless USB devices involves:

(a) a second gadget here, discovery mechanism TBD, but likely needing separate "register/unregister WUSB gadget" calls;

(b) updates to usb_gadget to include flags "is it wireless", "is it wired", plus (presumably in a wrapper structure) bandgroup and PHY info;

(c) presumably a wireless_ep wrapping a usb_ep, and reporting wireless-specific parameters like maxburst and maxsequence;

(d) configurations that are specific to wireless links;

(e) function drivers that understand wireless configs and will support wireless for (additional) function instances;

(f) a function to support association setup (like CBAF), not necessarily requiring a wireless adapter;

(g) composite device setup that can create one or more wireless configs, including appropriate association setup support;

(h) more, TBD.

int **config_ep_by_speed_and_alt**(struct usb_gadget * g, struct usb_function * f, struct usb_ep * _ep, u8 alt)
configures the given endpoint according to gadget speed.

Parameters

struct usb_gadget * g pointer to the gadget

struct usb_function * f usb function

struct usb_ep * _ep the endpoint to configure

u8 alt alternate setting number

Return

error code, 0 on success

Description

This function chooses the right descriptors for a given endpoint according to gadget speed and saves it in the endpoint desc field. If the endpoint already has a descriptor assigned to it - overwrites it with currently corresponding descriptor. The endpoint maxpacket field is updated according to the chosen descriptor.

Note

the supplied function should hold all the descriptors for supported speeds

int **config_ep_by_speed**(struct usb_gadget * g, struct usb_function * f, struct usb_ep * _ep)
configures the given endpoint according to gadget speed.

Parameters

struct usb_gadget * g pointer to the gadget

struct usb_function * f usb function

struct usb_ep * _ep the endpoint to configure

Return

error code, 0 on success

Description

This function chooses the right descriptors for a given endpoint according to gadget speed and saves it in the endpoint desc field. If the endpoint already has a descriptor assigned to it - overwrites it with currently corresponding descriptor. The endpoint maxpacket field is updated according to the chosen descriptor.

Note

the supplied function should hold all the descriptors for supported speeds

int **usb_add_function**(struct usb_configuration * config, struct usb_function * function)
add a function to a configuration

Parameters

struct usb_configuration * config the configuration

struct usb_function * function the function being added

Context

single threaded during gadget setup

Description

After initialization, each configuration must have one or more functions added to it. Adding a function involves calling its **bind()** method to allocate resources such as interface and string identifiers and endpoints.

This function returns the value of the function's bind(), which is zero for success else a negative errno value.

int **usb_function_deactivate**(struct usb_function * function)
prevent function and gadget enumeration

Parameters

struct usb_function * function the function that isn't yet ready to respond

Description

Blocks response of the gadget driver to host enumeration by preventing the data line pullup from being activated. This is normally called during **bind()** processing to change from the initial "ready to respond" state, or when a required resource becomes available.

For example, drivers that serve as a passthrough to a userspace daemon can block enumeration unless that daemon (such as an OBEX, MTP, or print server) is ready to handle host requests.

Not all systems support software control of their USB peripheral data pullups.

Returns zero on success, else negative errno.

int **usb_function_activate**(struct usb_function * function)
allow function and gadget enumeration

Parameters

struct usb_function * function function on which `usb_function_activate()` was called

Description

Reverses effect of `usb_function_deactivate()`. If no more functions are delaying their activation, the gadget driver will respond to host enumeration procedures.

Returns zero on success, else negative errno.

int **usb_interface_id**(struct usb_configuration * config, struct usb_function * function)
allocate an unused interface ID

Parameters

struct usb_configuration * config configuration associated with the interface

struct usb_function * function function handling the interface

Context

single threaded during gadget setup

Description

`usb_interface_id()` is called from usb_function.bind() callbacks to allocate new interface IDs. The function driver will then store that ID in interface, association, CDC union, and other descriptors. It will also handle any control requests targeted at that interface, particularly changing its altsetting via set_alt(). There may also be class-specific or vendor-specific requests to handle.

All interface identifier should be allocated using this routine, to ensure that for example different functions don't wrongly assign different meanings to the same identifier. Note that since interface identifiers are configuration-specific, functions used in more than one configuration (or more than once in a given configuration) need multiple versions of the relevant descriptors.

Returns the interface ID which was allocated; or -ENODEV if no more interface IDs can be allocated.

int **usb_add_config**(struct usb_composite_dev * cdev, struct usb_configuration * config, int (*bind)(struct usb_configuration *))
add a configuration to a device.

Parameters

struct usb_composite_dev * cdev wraps the USB gadget

struct usb_configuration * config the configuration, with bConfigurationValue assigned

int (*)(struct usb_configuration *) bind the configuration's bind function

Context

single threaded during gadget setup

Description

One of the main tasks of a composite **bind()** routine is to add each of the configurations it supports, using this routine.

This function returns the value of the configuration's **bind()**, which is zero for success else a negative errno value. Binding configurations assigns global resources including string IDs, and per-configuration resources such as interface IDs and endpoints.

int **usb_string_id**(struct usb_composite_dev * cdev)
allocate an unused string ID

Parameters

struct usb_composite_dev * cdev the device whose string descriptor IDs are being allocated

Context

single threaded during gadget setup

Description

usb_string_id() is called from bind() callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then store that ID in the appropriate descriptors and string table.

All string identifier should be allocated using this, **usb_string_ids_tab()** or **usb_string_ids_n()** routine, to ensure that for example different functions don' t wrongly assign different meanings to the same identifier.

int **usb_string_ids_tab**(struct usb_composite_dev * cdev, struct usb_string * str)
allocate unused string IDs in batch

Parameters

struct usb_composite_dev * cdev the device whose string descriptor IDs are being allocated

struct usb_string * str an array of usb_string objects to assign numbers to

Context

single threaded during gadget setup

Description

usb_string_ids() is called from bind() callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then copy IDs from the string table to the appropriate descriptors and string table for other languages.

All string identifier should be allocated using this, **usb_string_id()** or **usb_string_ids_n()** routine, to ensure that for example different functions don' t wrongly assign different meanings to the same identifier.

struct usb_string * **usb_gstrings_attach**(struct usb_composite_dev * cdev, struct usb_gadget_strings ** sp, unsigned n_strings)
attach gadget strings to a cdev and assign ids

Parameters

struct usb_composite_dev * cdev the device whose string descriptor IDs are being allocated and attached.

struct usb_gadget_strings ** sp an array of usb_gadget_strings to attach.

unsigned n_strings number of entries in each usb_strings array (sp[]->strings)

Description

This function will create a deep copy of usb_gadget_strings and usb_string and attach it to the cdev. The actual string (usb_string.s) will not be copied but only a referenced will be made. The struct usb_gadget_strings array may contain multiple languages and should be NULL terminated. The ->language pointer of each struct usb_gadget_strings has to contain the same amount of entries. For instance: sp[0] is en-US, sp[1] is es-ES. It is expected that the first usb_string entry of es-ES contains the translation of the first usb_string entry of en-US. Therefore both entries become the same id assign.

int **usb_string_ids_n**(struct usb_composite_dev * c, unsigned n)
allocate unused string IDs in batch

Parameters

struct usb_composite_dev * c the device whose string descriptor IDs are being allocated

unsigned n number of string IDs to allocate

Context

single threaded during gadget setup

Description

Returns the first requested ID. This ID and next **n**-1 IDs are now valid IDs. At least provided that **n** is non-zero because if it is, returns last requested ID which is now very useful information.

usb_string_ids_n() is called from bind() callbacks to allocate string IDs. Drivers for functions, configurations, or gadgets will then store that ID in the appropriate descriptors and string table.

All string identifier should be allocated using this, **usb_string_id()** or **usb_string_ids_n()** routine, to ensure that for example different functions don' t wrongly assign different meanings to the same identifier.

int **usb_composite_probe**(struct usb_composite_driver * driver)
register a composite driver

Parameters

struct usb_composite_driver * driver the driver to register

Context

single threaded during gadget setup

Description

This function is used to register drivers using the composite driver framework. The return value is zero, or a negative errno value. Those values normally come from the driver' s **bind** method, which does all the work of setting up the driver to match the hardware.

On successful return, the gadget is ready to respond to requests from the host, unless one of its components invokes usb_gadget_disconnect() while it was binding. That would usually be done in order to wait for some userspace participation.

void **usb_composite_unregister**(struct usb_composite_driver * driver)
unregister a composite driver

Parameters

struct usb_composite_driver * driver the driver to unregister

Description

This function is used to unregister drivers using the composite driver framework.

void **usb_composite_setup_continue**(struct usb_composite_dev * cdev)
Continue with the control transfer

Parameters

struct usb_composite_dev * cdev the composite device who's control transfer was kept waiting

Description

This function must be called by the USB function driver to continue with the control transfer's data/status stage in case it had requested to delay the data/status stages. A USB function's setup handler (e.g. set_alt()) can request the composite framework to delay the setup request's data/status stages by returning USB_GADGET_DELAYED_STATUS.

Composite Device Functions

At this writing, a few of the current gadget drivers have been converted to this framework. Near-term plans include converting all of them, except for `gadgetfs`.

20.2.4 Peripheral Controller Drivers

The first hardware supporting this API was the NetChip 2280 controller, which supports USB 2.0 high speed and is based on PCI. This is the `net2280` driver module. The driver supports Linux kernel versions 2.4 and 2.6; contact NetChip Technologies for development boards and product information.

Other hardware working in the `gadget` framework includes: Intel's PXA 25x and IXP42x series processors (`pxa2xx_udc`), Toshiba TC86c001 "Goku-S" (`goku_udc`), Renesas SH7705/7727 (`sh_udc`), MediaQ 11xx (`mq11xx_udc`), Hynix HMS30C7202 (`h7202_udc`), National 9303/4 (`n9604_udc`), Texas Instruments OMAP (`omap_udc`), Sharp LH7A40x (`lh7a40x_udc`), and more. Most of those are full speed controllers.

At this writing, there are people at work on drivers in this framework for several other USB device controllers, with plans to make many of them be widely available.

A partial USB simulator, the `dummy_hcd` driver, is available. It can act like a net2280, a pxa25x, or an sa11x0 in terms of available endpoints and device speeds; and it simulates control, bulk, and to some extent interrupt transfers. That lets you develop some parts of a gadget driver on a normal PC, without any special hardware, and perhaps with the assistance of tools such as GDB running with User Mode Linux. At least one person has expressed interest in adapting that approach, hooking it up to a simulator for a microcontroller. Such simulators can help debug subsystems where the runtime hardware is unfriendly to software development, or is not yet available.

Support for other controllers is expected to be developed and contributed over time, as this driver framework evolves.

20.2.5 Gadget Drivers

In addition to Gadget Zero (used primarily for testing and development with drivers for usb controller hardware), other gadget drivers exist.

There's an `ethernet` gadget driver, which implements one of the most useful Communications Device Class (CDC) models. One of the standards for cable modem interoperability even specifies the use of this ethernet model as one of two mandatory options. Gadgets using this code look to a USB host as if they're an Ethernet adapter. It provides access to a network where the gadget's CPU is one host, which could easily be bridging, routing, or firewalling access to other networks. Since some hardware can't fully implement the CDC Ethernet requirements, this driver also implements a "good parts only" subset of CDC Ethernet. (That subset doesn't advertise itself as CDC Ethernet, to avoid creating problems.)

Support for Microsoft's `RNDIS` protocol has been contributed by Pengutronix and Auerswald GmbH. This is like CDC Ethernet, but it runs on more slightly USB hardware (but less than the CDC subset). However, its main claim to fame is being able to connect directly to recent versions of Windows, using drivers that Microsoft bundles and supports, making it much simpler to network with Windows.

There is also support for user mode gadget drivers, using `gadgetfs`. This provides a User Mode API that presents each endpoint as a single file descriptor. I/O is done using normal `read()` and `read()` calls. Familiar tools like GDB and pthreads can be used to develop and debug user mode drivers, so that once a robust controller driver is available many applications for it won't require new kernel mode software. Linux 2.6 Async I/O (AIO) support is available, so that user mode software can stream data with only slightly more overhead than a kernel driver.

There's a USB Mass Storage class driver, which provides a different solution for interoperability with systems such as MS-Windows and MacOS. That Mass Storage driver uses a file or block device as backing store for a drive, like the `loop` driver. The USB host uses the BBB, CB, or CBI versions of the mass storage class specification, using transparent SCSI commands to access the data from the backing store.

There's a "serial line" driver, useful for TTY style operation over USB. The latest version of that driver supports CDC ACM style operation, like a USB modem, and so on most hardware it can interoperate easily with MS-Windows. One interesting use of that driver is in boot firmware (like a BIOS), which can sometimes use that model with very small systems without real serial lines.

Support for other kinds of gadget is expected to be developed and contributed over time, as this driver framework evolves.

20.2.6 USB On-The-GO (OTG)

USB OTG support on Linux 2.6 was initially developed by Texas Instruments for OMAP 16xx and 17xx series processors. Other OTG systems should work in similar ways, but the hardware level details could be very different.

Systems need specialized hardware support to implement OTG, notably including a special Mini-AB jack and associated transceiver to support Dual-Role operation: they can act either as a host, using the standard Linux-USB host side driver stack, or as a peripheral, using this `gadget` framework. To do that, the system software relies on small additions to those programming interfaces, and on a new internal component (here called an "OTG Controller") affecting which driver stack connects to the OTG port. In each role, the system can re-use the existing pool of hardware-neutral drivers, layered on top of the controller driver interfaces (`usb_bus` or `usb_gadget`). Such drivers need at most minor changes, and most of the calls added to support OTG can also benefit non-OTG products.

- Gadget drivers test the `is_otg` flag, and use it to determine whether or not to include an OTG descriptor in each of their configurations.
- Gadget drivers may need changes to support the two new OTG protocols, exposed in new gadget attributes such as `b_hnp_enable` flag. HNP support should be reported through a user interface (two LEDs could suffice), and is triggered in some cases when the host suspends the peripheral. SRP support can be user-initiated just like remote wakeup, probably by pressing the same button.
- On the host side, USB device drivers need to be taught to trigger HNP at appropriate moments, using `usb_suspend_device()`. That also conserves battery power, which is useful even for non-OTG configurations.
- Also on the host side, a driver must support the OTG "Targeted Peripheral List". That's just a whitelist, used to reject peripherals not supported with a given Linux OTG host. This whitelist is product-specific; each product must modify `otg_whitelist.h` to match its interoperability specification.

  Non-OTG Linux hosts, like PCs and workstations, normally have some solution for adding drivers, so that peripherals that aren't recognized can eventually be supported. That approach is unreasonable for consumer products that may never have their firmware upgraded, and where it's usually unrealistic to expect traditional PC/workstation/server kinds of support model to work. For example, it's often impractical to change device firmware once the product has been distributed, so driver bugs can't normally be fixed if they're found after shipment.

Additional changes are needed below those hardware-neutral `usb_bus` and `usb_gadget` driver interfaces; those aren't discussed here in any detail. Those affect the hardware-specific code for each USB Host or Peripheral controller, and how the HCD initializes (since OTG can be active only on a single port). They also involve what may be called an OTG Controller Driver, managing the OTG transceiver and the OTG state machine logic as well as much of the root hub behavior for the OTG port. The OTG controller driver needs to activate and deactivate USB controllers depending on the relevant device role. Some related changes were needed inside usbcore, so that it can identify OTG-capable devices and respond appropriately to HNP or SRP protocols.

20.3 USB Anchors

20.3.1 What is anchor?

A USB driver needs to support some callbacks requiring a driver to cease all IO to an interface. To do so, a driver has to keep track of the URBs it has submitted to know they' ve all completed or to call usb_kill_urb for them. The anchor is a data structure takes care of keeping track of URBs and provides methods to deal with multiple URBs.

20.3.2 Allocation and Initialisation

There' s no API to allocate an anchor. It is simply declared as struct usb_anchor. `init_usb_anchor()` must be called to initialise the data structure.

20.3.3 Deallocation

Once it has no more URBs associated with it, the anchor can be freed with normal memory management operations.

20.3.4 Association and disassociation of URBs with anchors

An association of URBs to an anchor is made by an explicit call to `usb_anchor_urb()`. The association is maintained until an URB is finished by (successful) completion. Thus disassociation is automatic. A function is provided to forcibly finish (kill) all URBs associated with an anchor. Furthermore, disassociation can be made with `usb_unanchor_urb()`

20.3.5 Operations on multitudes of URBs

`usb_kill_anchored_urbs()`

This function kills all URBs associated with an anchor. The URBs are called in the reverse temporal order they were submitted. This way no data can be reordered.

`usb_unlink_anchored_urbs()`

This function unlinks all URBs associated with an anchor. The URBs are processed in the reverse temporal order they were submitted. This is similar to `usb_kill_anchored_urbs()`, but it will not sleep. Therefore no guarantee is made that the URBs have been unlinked when the call returns. They may be unlinked later but will be unlinked in finite time.

usb_scuttle_anchored_urbs()

All URBs of an anchor are unanchored en masse.

usb_wait_anchor_empty_timeout()

This function waits for all URBs associated with an anchor to finish or a timeout, whichever comes first. Its return value will tell you whether the timeout was reached.

usb_anchor_empty()

Returns true if no URBs are associated with an anchor. Locking is the caller's responsibility.

usb_get_from_anchor()

Returns the oldest anchored URB of an anchor. The URB is unanchored and returned with a reference. As you may mix URBs to several destinations in one anchor you have no guarantee the chronologically first submitted URB is returned.

20.4 USB bulk streams

20.4.1 Background

Bulk endpoint streams were added in the USB 3.0 specification. Streams allow a device driver to overload a bulk endpoint so that multiple transfers can be queued at once.

Streams are defined in sections 4.4.6.4 and 8.12.1.4 of the Universal Serial Bus 3.0 specification at https://www.usb.org/developers/docs/ The USB Attached SCSI Protocol, which uses streams to queue multiple SCSI commands, can be found on the T10 website (https://t10.org/).

20.4.2 Device-side implications

Once a buffer has been queued to a stream ring, the device is notified (through an out-of-band mechanism on another endpoint) that data is ready for that stream ID. The device then tells the host which "stream" it wants to start. The host can also initiate a transfer on a stream without the device asking, but the device can refuse that transfer. Devices can switch between streams at any time.

20.4.3 Driver implications

```
int usb_alloc_streams(struct usb_interface *interface,
              struct usb_host_endpoint **eps, unsigned int num_eps,
              unsigned int num_streams, gfp_t mem_flags);
```

Device drivers will call this API to request that the host controller driver allocate memory so the driver can use up to num_streams stream IDs. They must pass an array of usb_host_endpoints that need to be setup with similar stream IDs. This is to ensure that a UASP driver will be able to use the same stream ID for the bulk IN and OUT endpoints used in a Bi-directional command sequence.

The return value is an error condition (if one of the endpoints doesn't support streams, or the xHCI driver ran out of memory), or the number of streams the host controller allocated for this endpoint. The xHCI host controller hardware declares how many stream IDs it can support, and each bulk endpoint on a SuperSpeed device will say how many stream IDs it can handle. Therefore, drivers should be able to deal with being allocated less stream IDs than they requested.

Do NOT call this function if you have URBs enqueued for any of the endpoints passed in as arguments. Do not call this function to request less than two streams.

Drivers will only be allowed to call this API once for the same endpoint without calling `usb_free_streams()`. This is a simplification for the xHCI host controller driver, and may change in the future.

20.4.4 Picking new Stream IDs to use

Stream ID 0 is reserved, and should not be used to communicate with devices. If `usb_alloc_streams()` returns with a value of N, you may use streams 1 though N. To queue an URB for a specific stream, set the urb->stream_id value. If the endpoint does not support streams, an error will be returned.

Note that new API to choose the next stream ID will have to be added if the xHCI driver supports secondary stream IDs.

20.4.5 Clean up

If a driver wishes to stop using streams to communicate with the device, it should call:

```
void usb_free_streams(struct usb_interface *interface,
              struct usb_host_endpoint **eps, unsigned int num_eps,
              gfp_t mem_flags);
```

All stream IDs will be deallocated when the driver releases the interface, to ensure that drivers that don't support streams will be able to use the endpoint.

20.5 USB core callbacks

20.5.1 What callbacks will usbcore do?

Usbcore will call into a driver through callbacks defined in the driver structure and through the completion handler of URBs a driver submits. Only the former are in the scope of this document. These two kinds of callbacks are completely independent of each other. Information on the completion callback can be found in USB Request Block (URB).

The callbacks defined in the driver structure are:

1. Hotplugging callbacks:

- **@probe:** Called to see if the driver is willing to manage a particular interface on a device.
- **@disconnect:** Called when the interface is no longer accessible, usually because its device has been (or is being) disconnected or the driver module is being unloaded.

2. Odd backdoor through usbfs:

- **@ioctl:** Used for drivers that want to talk to userspace through the "usbfs" filesystem. This lets devices provide ways to expose information to user space regardless of where they do (or don't) show up otherwise in the filesystem.

3. Power management (PM) callbacks:

- **@suspend:** Called when the device is going to be suspended.
- **@resume:** Called when the device is being resumed.
- **@reset_resume:** Called when the suspended device has been reset instead of being resumed.

4. Device level operations:

- **@pre_reset:** Called when the device is about to be reset.
- **@post_reset:** Called after the device has been reset

The ioctl interface (2) should be used only if you have a very good reason. Sysfs is preferred these days. The PM callbacks are covered separately in Power Management for USB.

20.5.2 Calling conventions

All callbacks are mutually exclusive. There's no need for locking against other USB callbacks. All callbacks are called from a task context. You may sleep. However, it is important that all sleeps have a small fixed upper limit in time. In particular you must not call out to user space and await results.

20.5.3 Hotplugging callbacks

These callbacks are intended to associate and disassociate a driver with an interface. A driver's bond to an interface is exclusive.

The probe() callback

```
int (*probe) (struct usb_interface *intf,
              const struct usb_device_id *id);
```

Accept or decline an interface. If you accept the device return 0, otherwise -ENODEV or -ENXIO. Other error codes should be used only if a genuine error occurred during initialisation which prevented a driver from accepting a device that would else have been accepted. You are strongly encouraged to use usbcore's facility, usb_set_intfdata(), to associate a data structure with an interface, so that you know which internal state and identity you associate with a particular interface. The device will not be suspended and you may do IO to the interface you are called for and endpoint 0 of the device. Device initialisation that doesn't take too long is a good idea here.

The disconnect() callback

```
void (*disconnect) (struct usb_interface *intf);
```

This callback is a signal to break any connection with an interface. You are not allowed any IO to a device after returning from this callback. You also may not do any other operation that may interfere with another driver bound the interface, eg. a power management operation. If you are called due to a physical disconnection, all your URBs will be killed by usbcore. Note that in this case disconnect will be called some time after the physical disconnection. Thus your driver must be prepared to deal with failing IO even prior to the callback.

20.5.4 Device level callbacks

pre_reset

```
int (*pre_reset)(struct usb_interface *intf);
```

A driver or user space is triggering a reset on the device which contains the interface passed as an argument. Cease IO, wait for all outstanding URBs to complete, and save any device state you need to restore. No more URBs may be submitted until the post_reset method is called.

If you need to allocate memory here, use GFP_NOIO or GFP_ATOMIC, if you are in atomic context.

post_reset

```
int (*post_reset)(struct usb_interface *intf);
```

The reset has completed. Restore any saved device state and begin using the device again.

If you need to allocate memory here, use GFP_NOIO or GFP_ATOMIC, if you are in atomic context.

20.5.5 Call sequences

No callbacks other than probe will be invoked for an interface that isn't bound to your driver.

Probe will never be called for an interface bound to a driver. Hence following a successful probe, disconnect will be called before there is another probe for the same interface.

Once your driver is bound to an interface, disconnect can be called at any time except in between pre_reset and post_reset. pre_reset is always followed by post_reset, even if the reset failed or the device has been unplugged.

suspend is always followed by one of: resume, reset_resume, or disconnect.

20.6 USB DMA

In Linux 2.5 kernels (and later), USB device drivers have additional control over how DMA may be used to perform I/O operations. The APIs are detailed in the kernel usb programming guide (kerneldoc, from the source code).

20.6.1 API overview

The big picture is that USB drivers can continue to ignore most DMA issues, though they still must provide DMA-ready buffers (see `Documentation/DMA-API-HOWTO.txt`). That's how they've worked through the 2.4 (and earlier) kernels, or they can now be DMA-aware.

DMA-aware usb drivers:

- New calls enable DMA-aware drivers, letting them allocate dma buffers and manage dma mappings for existing dma-ready buffers (see below).
- URBs have an additional "transfer_dma" field, as well as a transfer_flags bit saying if it's valid. (Control requests also have "setup_dma", but drivers must not use it.)
- "usbcore" will map this DMA address, if a DMA-aware driver didn't do it first and set `URB_NO_TRANSFER_DMA_MAP`. HCDs don't manage dma mappings for URBs.

- There's a new "generic DMA API", parts of which are usable by USB device drivers. Never use dma_set_mask() on any USB interface or device; that would potentially break all devices sharing that bus.

20.6.2 Eliminating copies

It's good to avoid making CPUs copy data needlessly. The costs can add up, and effects like cache-trashing can impose subtle penalties.

- If you're doing lots of small data transfers from the same buffer all the time, that can really burn up resources on systems which use an IOMMU to manage the DMA mappings. It can cost MUCH more to set up and tear down the IOMMU mappings with each request than perform the I/O!

  For those specific cases, USB has primitives to allocate less expensive memory. They work like kmalloc and kfree versions that give you the right kind of addresses to store in urb->transfer_buffer and urb->transfer_dma. You'd also set `URB_NO_TRANSFER_DMA_MAP` in urb->transfer_flags:

```
void *usb_alloc_coherent (struct usb_device *dev, size_t size,
        int mem_flags, dma_addr_t *dma);

void usb_free_coherent (struct usb_device *dev, size_t size,
        void *addr, dma_addr_t dma);
```

  Most drivers should **NOT** be using these primitives; they don't need to use this type of memory ("dma-coherent"), and memory returned from `kmalloc()` will work just fine.

  The memory buffer returned is "dma-coherent"; sometimes you might need to force a consistent memory access ordering by using memory barriers. It's not using a streaming DMA mapping, so it's good for small transfers on systems where the I/O would otherwise thrash an IOMMU mapping. (See `Documentation/DMA-API-HOWTO.txt` for definitions of "coherent" and "streaming" DMA mappings.)

  Asking for 1/Nth of a page (as well as asking for N pages) is reasonably space-efficient.

  On most systems the memory returned will be uncached, because the semantics of dma-coherent memory require either bypassing CPU caches or using cache hardware with bus-snooping support. While x86 hardware has such bus-snooping, many other systems use software to flush cache lines to prevent DMA conflicts.

- Devices on some EHCI controllers could handle DMA to/from high memory.

  Unfortunately, the current Linux DMA infrastructure doesn't have a sane way to expose these capabilities ⋯and in any case, HIGHMEM is mostly a design wart specific to x86_32. So your best bet is to ensure you never pass a highmem buffer into a USB driver. That's easy; it's the default behavior. Just don't override it; e.g. with `NETIF_F_HIGHDMA`.

  This may force your callers to do some bounce buffering, copying from high memory to "normal" DMA memory. If you can come up with a good way to

fix this issue (for x86_32 machines with over 1 GByte of memory), feel free to submit patches.

20.6.3 Working with existing buffers

Existing buffers aren't usable for DMA without first being mapped into the DMA address space of the device. However, most buffers passed to your driver can safely be used with such DMA mapping. (See the first section of Documentation/DMA-API-HOWTO.txt, titled "What memory is DMA-able?")

- When you're using scatterlists, you can map everything at once. On some systems, this kicks in an IOMMU and turns the scatterlists into single DMA transactions:

```
int usb_buffer_map_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int nents);

void usb_buffer_dmasync_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int n_hw_ents);

void usb_buffer_unmap_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int n_hw_ents);
```

  It's probably easier to use the new `usb_sg_*()` calls, which do the DMA mapping and apply other tweaks to make scatterlist i/o be fast.

- Some drivers may prefer to work with the model that they're mapping large buffers, synchronizing their safe re-use. (If there's no re-use, then let usbcore do the map/unmap.) Large periodic transfers make good examples here, since it's cheaper to just synchronize the buffer than to unmap it each time an urb completes and then re-map it on during resubmission.

  These calls all work with initialized urbs: `urb->dev`, `urb->pipe`, `urb->transfer_buffer`, and `urb->transfer_buffer_length` must all be valid when these calls are used (`urb->setup_packet` must be valid too if urb is a control request):

```
struct urb *usb_buffer_map (struct urb *urb);

void usb_buffer_dmasync (struct urb *urb);

void usb_buffer_unmap (struct urb *urb);
```

  The calls manage `urb->transfer_dma` for you, and set `URB_NO_TRANSFER_DMA_MAP` so that usbcore won't map or unmap the buffer. They cannot be used for setup_packet buffers in control requests.

Note that several of those interfaces are currently commented out, since they don't have current users. See the source code. Other than the dmasync calls (where the underlying DMA primitives have changed), most of them can easily be commented back in if you want to use them.

20.7 USB Request Block (URB)

Revised 2000-Dec-05

Again 2002-Jul-06

Again 2005-Sep-19

Again 2017-Mar-29

Note: The USB subsystem now has a substantial section at The Linux-USB Host Side API section, generated from the current source code. This particular documentation file isn't complete and may not be updated to the last version; don't rely on it except for a quick overview.

20.7.1 Basic concept or 'What is an URB?'

The basic idea of the new driver is message passing, the message itself is called USB Request Block, or URB for short.

- An URB consists of all relevant information to execute any USB transaction and deliver the data and status back.
- Execution of an URB is inherently an asynchronous operation, i.e. the `usb_submit_urb()` call returns immediately after it has successfully queued the requested action.
- Transfers for one URB can be canceled with `usb_unlink_urb()` at any time.
- Each URB has a completion handler, which is called after the action has been successfully completed or canceled. The URB also contains a context-pointer for passing information to the completion handler.
- Each endpoint for a device logically supports a queue of requests. You can fill that queue, so that the USB hardware can still transfer data to an endpoint while your driver handles completion of another. This maximizes use of USB bandwidth, and supports seamless streaming of data to (or from) devices when using periodic transfer modes.

20.7.2 The URB structure

Some of the fields in struct `urb` are:

```
struct urb
{
// (IN) device and pipe specify the endpoint queue
      struct usb_device *dev;         // pointer to associated USB device
      unsigned int pipe;              // endpoint information

      unsigned int transfer_flags;    // URB_ISO_ASAP, URB_SHORT_NOT_OK,␣
↪etc.
```

(continues on next page)

(continued from previous page)

```
// (IN) all urbs need completion routines
      void *context;                  // context for completion routine
      usb_complete_t complete;        // pointer to completion routine

// (OUT) status after each completion
      int status;                     // returned status

// (IN) buffer used for data transfers
      void *transfer_buffer;          // associated data buffer
      u32 transfer_buffer_length;     // data buffer length
      int number_of_packets;          // size of iso_frame_desc

// (OUT) sometimes only part of CTRL/BULK/INTR transfer_buffer is used
      u32 actual_length;              // actual data buffer length

// (IN) setup stage for CTRL (pass a struct usb_ctrlrequest)
      unsigned char *setup_packet;    // setup packet (control only)

// Only for PERIODIC transfers (ISO, INTERRUPT)
  // (IN/OUT) start_frame is set unless URB_ISO_ASAP isn't set
      int start_frame;                // start frame
      int interval;                   // polling interval

  // ISO only: packets are only "best effort"; each can have errors
      int error_count;                // number of errors
      struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

Your driver must create the “pipe”value using values from the appropriate endpoint descriptor in an interface that it’s claimed.

20.7.3 How to get an URB?

URBs are allocated by calling `usb_alloc_urb()`:

```
struct urb *usb_alloc_urb(int isoframes, int mem_flags)
```

Return value is a pointer to the allocated URB, 0 if allocation failed. The parameter isoframes specifies the number of isochronous transfer frames you want to schedule. For CTRL/BULK/INT, use 0. The mem_flags parameter holds standard memory allocation flags, letting you control (among other things) whether the underlying code may block or not.

To free an URB, use `usb_free_urb()`:

```
void usb_free_urb(struct urb *urb)
```

You may free an urb that you’ve submitted, but which hasn’t yet been returned to you in a completion callback. It will automatically be deallocated when it is no longer in use.

20.7.4 What has to be filled in?

Depending on the type of transaction, there are some inline functions defined in `linux/usb.h` to simplify the initialization, such as `usb_fill_control_urb()`, `usb_fill_bulk_urb()` and `usb_fill_int_urb()`. In general, they need the usb device pointer, the pipe (usual format from usb.h), the transfer buffer, the desired transfer length, the completion handler, and its context. Take a look at the some existing drivers to see how they' re used.

Flags:

- For ISO there are two startup behaviors: Specified start_frame or ASAP.
- For ASAP set `URB_ISO_ASAP` in transfer_flags.

If short packets should NOT be tolerated, set `URB_SHORT_NOT_OK` in transfer_flags.

20.7.5 How to submit an URB?

Just call `usb_submit_urb()`:

```
int usb_submit_urb(struct urb *urb, int mem_flags)
```

The `mem_flags` parameter, such as `GFP_ATOMIC`, controls memory allocation, such as whether the lower levels may block when memory is tight.

It immediately returns, either with status 0 (request queued) or some error code, usually caused by the following:

- Out of memory (`-ENOMEM`)
- Unplugged device (`-ENODEV`)
- Stalled endpoint (`-EPIPE`)
- Too many queued ISO transfers (`-EAGAIN`)
- Too many requested ISO frames (`-EFBIG`)
- Invalid INT interval (`-EINVAL`)
- More than one packet for INT (`-EINVAL`)

After submission, `urb->status` is `-EINPROGRESS`; however, you should never look at that value except in your completion callback.

For isochronous endpoints, your completion handlers should (re)submit URBs to the same endpoint with the `URB_ISO_ASAP` flag, using multi-buffering, to get seamless ISO streaming.

20.7.6 How to cancel an already running URB?

There are two ways to cancel an URB you’ ve submitted but which hasn’ t been returned to your driver yet. For an asynchronous cancel, call `usb_unlink_urb()`:

```
int usb_unlink_urb(struct urb *urb)
```

It removes the urb from the internal list and frees all allocated HW descriptors. The status is changed to reflect unlinking. Note that the URB will not normally have finished when `usb_unlink_urb()` returns; you must still wait for the completion handler to be called.

To cancel an URB synchronously, call `usb_kill_urb()`:

```
void usb_kill_urb(struct urb *urb)
```

It does everything `usb_unlink_urb()` does, and in addition it waits until after the URB has been returned and the completion handler has finished. It also marks the URB as temporarily unusable, so that if the completion handler or anyone else tries to resubmit it they will get a `-EPERM` error. Thus you can be sure that when `usb_kill_urb()` returns, the URB is totally idle.

There is a lifetime issue to consider. An URB may complete at any time, and the completion handler may free the URB. If this happens while `usb_unlink_urb()` or `usb_kill_urb()` is running, it will cause a memory-access violation. The driver is responsible for avoiding this, which often means some sort of lock will be needed to prevent the URB from being deallocated while it is still in use.

On the other hand, since usb_unlink_urb may end up calling the completion handler, the handler must not take any lock that is held when usb_unlink_urb is invoked. The general solution to this problem is to increment the URB’ s reference count while holding the lock, then drop the lock and call usb_unlink_urb or usb_kill_urb, and then decrement the URB’ s reference count. You increment the reference count by calling :c:func`usb_get_urb`:

```
struct urb *usb_get_urb(struct urb *urb)
```

(ignore the return value; it is the same as the argument) and decrement the reference count by calling `usb_free_urb()`. Of course, none of this is necessary if there’ s no danger of the URB being freed by the completion handler.

20.7.7 What about the completion handler?

The handler is of the following type:

```
typedef void (*usb_complete_t)(struct urb *)
```

I.e., it gets the URB that caused the completion call. In the completion handler, you should have a look at `urb->status` to detect any USB errors. Since the context parameter is included in the URB, you can pass information to the completion handler.

Note that even when an error (or unlink) is reported, data may have been transferred. That’s because USB transfers are packetized; it might take sixteen packets

to transfer your 1KByte buffer, and ten of them might have transferred successfully before the completion was called.

> **Warning:** NEVER SLEEP IN A COMPLETION HANDLER.
>
> These are often called in atomic context.

In the current kernel, completion handlers run with local interrupts disabled, but in the future this will be changed, so don't assume that local IRQs are always disabled inside completion handlers.

20.7.8 How to do isochronous (ISO) transfers?

Besides the fields present on a bulk transfer, for ISO, you also also have to set `urb->interval` to say how often to make transfers; it's often one per frame (which is once every microframe for highspeed devices). The actual interval used will be a power of two that's no bigger than what you specify. You can use the `usb_fill_int_urb()` macro to fill most ISO transfer fields.

For ISO transfers you also have to fill a `usb_iso_packet_descriptor` structure, allocated at the end of the URB by `usb_alloc_urb()`, for each packet you want to schedule.

The `usb_submit_urb()` call modifies `urb->interval` to the implemented interval value that is less than or equal to the requested interval value. If `URB_ISO_ASAP` scheduling is used, `urb->start_frame` is also updated.

For each entry you have to specify the data offset for this frame (base is transfer_buffer), and the length you want to write/expect to read. After completion, actual_length contains the actual transferred length and status contains the resulting status for the ISO transfer for this frame. It is allowed to specify a varying length from frame to frame (e.g. for audio synchronisation/adaptive transfer rates). You can also use the length 0 to omit one or more frames (striping).

For scheduling you can choose your own start frame or `URB_ISO_ASAP`. As explained earlier, if you always keep at least one URB queued and your completion keeps (re)submitting a later URB, you'll get smooth ISO streaming (if usb bandwidth utilization allows).

If you specify your own start frame, make sure it's several frames in advance of the current frame. You might want this model if you're synchronizing ISO data with some other event stream.

20.7.9 How to start interrupt (INT) transfers?

Interrupt transfers, like isochronous transfers, are periodic, and happen in intervals that are powers of two (1, 2, 4 etc) units. Units are frames for full and low speed devices, and microframes for high speed ones. You can use the `usb_fill_int_urb()` macro to fill INT transfer fields.

The `usb_submit_urb()` call modifies `urb->interval` to the implemented interval value that is less than or equal to the requested interval value.

In Linux 2.6, unlike earlier versions, interrupt URBs are not automagically restarted when they complete. They end when the completion handler is called, just like other URBs. If you want an interrupt URB to be restarted, your completion handler must resubmit it. s

20.8 Power Management for USB

Author Alan Stern <stern@rowland.harvard.edu>

Date Last-updated: February 2014

20.8.1 What is Power Management?

Power Management (PM) is the practice of saving energy by suspending parts of a computer system when they aren't being used. While a component is `suspended` it is in a nonfunctional low-power state; it might even be turned off completely. A suspended component can be `resumed` (returned to a functional full-power state) when the kernel needs to use it. (There also are forms of PM in which components are placed in a less functional but still usable state instead of being suspended; an example would be reducing the CPU's clock rate. This document will not discuss those other forms.)

When the parts being suspended include the CPU and most of the rest of the system, we speak of it as a "system suspend". When a particular device is turned off while the system as a whole remains running, we call it a "dynamic suspend" (also known as a "runtime suspend" or "selective suspend"). This document concentrates mostly on how dynamic PM is implemented in the USB subsystem, although system PM is covered to some extent (see `Documentation/power/*.rst` for more information about system PM).

System PM support is present only if the kernel was built with `CONFIG_SUSPEND` or `CONFIG_HIBERNATION` enabled. Dynamic PM support

for USB is present whenever the kernel was built with `CONFIG_PM` enabled.

[Historically, dynamic PM support for USB was present only if the kernel had been built with `CONFIG_USB_SUSPEND` enabled (which depended on `CONFIG_PM_RUNTIME`). Starting with the 3.10 kernel release, dynamic PM support for USB was present whenever the kernel was built with `CONFIG_PM_RUNTIME` enabled. The `CONFIG_USB_SUSPEND` option had been eliminated.]

20.8.2 What is Remote Wakeup?

When a device has been suspended, it generally doesn't resume until the computer tells it to. Likewise, if the entire computer has been suspended, it generally doesn't resume until the user tells it to, say by pressing a power button or opening the cover.

However some devices have the capability of resuming by themselves, or asking the kernel to resume them, or even telling the entire computer to resume. This capability goes by several names such as "Wake On LAN"; we will refer to it generically as "remote wakeup". When a device is enabled for remote wakeup and it is suspended, it may resume itself (or send a request to be resumed) in response to some external event. Examples include a suspended keyboard resuming when a key is pressed, or a suspended USB hub resuming when a device is plugged in.

20.8.3 When is a USB device idle?

A device is idle whenever the kernel thinks it's not busy doing anything important and thus is a candidate for being suspended. The exact definition depends on the device's driver; drivers are allowed to declare that a device isn't idle even when there's no actual communication taking place. (For example, a hub isn't considered idle unless all the devices plugged into that hub are already suspended.) In addition, a device isn't considered idle so long as a program keeps its usbfs file open, whether or not any I/O is going on.

If a USB device has no driver, its usbfs file isn't open, and it isn't being accessed through sysfs, then it definitely is idle.

20.8.4 Forms of dynamic PM

Dynamic suspends occur when the kernel decides to suspend an idle device. This is called `autosuspend` for short. In general, a device won't be autosuspended unless it has been idle for some minimum period of time, the so-called idle-delay time.

Of course, nothing the kernel does on its own initiative should prevent the computer or its devices from working properly. If a device has been autosuspended and a program tries to use it, the kernel will automatically resume the device (autoresume). For the same reason, an autosuspended device will usually have remote wakeup enabled, if the device supports remote wakeup.

It is worth mentioning that many USB drivers don't support autosuspend. In fact, at the time of this writing (Linux 2.6.23) the only drivers which do support it are the hub driver, kaweth, asix, usblp, usblcd, and usb-skeleton (which doesn't count). If a non-supporting driver is bound to a device, the device won't be autosuspended. In effect, the kernel pretends the device is never idle.

We can categorize power management events in two broad classes: external and internal. External events are those triggered by some agent outside the USB stack: system suspend/resume (triggered by userspace), manual dynamic resume (also triggered by userspace), and remote wakeup (triggered by the device). Internal events are those triggered within the USB stack: autosuspend and autoresume.

Note that all dynamic suspend events are internal; external agents are not allowed to issue dynamic suspends.

20.8.5 The user interface for dynamic PM

The user interface for controlling dynamic PM is located in the `power/` subdirectory of each USB device's sysfs directory, that is, in `/sys/bus/usb/devices/.../power/` where "..." is the device's ID. The relevant attribute files are: wakeup, control, and `autosuspend_delay_ms`. (There may also be a file named `level`; this file was deprecated as of the 2.6.35 kernel and replaced by the `control` file. In 2.6.38 the `autosuspend` file will be deprecated and replaced by the `autosuspend_delay_ms` file. The only difference is that the newer file expresses the delay in milliseconds whereas the older file uses seconds. Confusingly, both files are present in 2.6.37 but only `autosuspend` works.)

`power/wakeup`

This file is empty if the device does not support remote wakeup. Otherwise the file contains either the word `enabled` or the word `disabled`, and you can write those words to the file. The setting determines whether or not remote wakeup will be enabled when the device is next suspended. (If the setting is changed while the device is suspended, the change won't take effect until the following suspend.)

`power/control`

This file contains one of two words: `on` or `auto`. You can write those words to the file to change the device's setting.

- `on` means that the device should be resumed and autosuspend is not allowed. (Of course, system suspends are still allowed.)
- `auto` is the normal state in which the kernel is allowed to autosuspend and autoresume the device.

(In kernels up to 2.6.32, you could also specify `suspend`, meaning that the device should remain suspended and autoresume was not allowed. This setting is no longer supported.)

`power/autosuspend_delay_ms`

This file contains an integer value, which is the number of milliseconds the device should remain idle before the kernel will autosuspend it (the idle-delay time). The default is 2000. 0 means to autosuspend as soon as the device becomes idle, and negative values mean never to autosuspend. You can write a number to the file to change the autosuspend idle-delay time.

Writing `-1` to `power/autosuspend_delay_ms` and writing `on` to `power/control` do essentially the same thing – they both prevent the device from being autosuspended. Yes, this is a redundancy in the API.

(In 2.6.21 writing `0` to `power/autosuspend` would prevent the device from being autosuspended; the behavior was changed in 2.6.22. The `power/autosuspend`

attribute did not exist prior to 2.6.21, and the `power/level` attribute did not exist prior to 2.6.22. `power/control` was added in 2.6.34, and `power/autosuspend_delay_ms` was added in 2.6.37 but did not become functional until 2.6.38.)

20.8.6 Changing the default idle-delay time

The default autosuspend idle-delay time (in seconds) is controlled by a module parameter in usbcore. You can specify the value when usbcore is loaded. For example, to set it to 5 seconds instead of 2 you would do:

```
modprobe usbcore autosuspend=5
```

Equivalently, you could add to a configuration file in /etc/modprobe.d a line saying:

```
options usbcore autosuspend=5
```

Some distributions load the usbcore module very early during the boot process, by means of a program or script running from an initramfs image. To alter the parameter value you would have to rebuild that image.

If usbcore is compiled into the kernel rather than built as a loadable module, you can add:

```
usbcore.autosuspend=5
```

to the kernel's boot command line.

Finally, the parameter value can be changed while the system is running. If you do:

```
echo 5 >/sys/module/usbcore/parameters/autosuspend
```

then each new USB device will have its autosuspend idle-delay initialized to 5. (The idle-delay values for already existing devices will not be affected.)

Setting the initial default idle-delay to -1 will prevent any autosuspend of any USB device. This has the benefit of allowing you then to enable autosuspend for selected devices.

20.8.7 Warnings

The USB specification states that all USB devices must support power management. Nevertheless, the sad fact is that many devices do not support it very well. You can suspend them all right, but when you try to resume them they disconnect themselves from the USB bus or they stop working entirely. This seems to be especially prevalent among printers and scanners, but plenty of other types of device have the same deficiency.

For this reason, by default the kernel disables autosuspend (the `power/control` attribute is initialized to `on`) for all devices other than hubs. Hubs, at least, appear to be reasonably well-behaved in this regard.

(In 2.6.21 and 2.6.22 this wasn't the case. Autosuspend was enabled by default for almost all USB devices. A number of people experienced problems as a result.)

This means that non-hub devices won't be autosuspended unless the user or a program explicitly enables it. As of this writing there aren't any widespread programs which will do this; we hope that in the near future device managers such as HAL will take on this added responsibility. In the meantime you can always carry out the necessary operations by hand or add them to a udev script. You can also change the idle-delay time; 2 seconds is not the best choice for every device.

If a driver knows that its device has proper suspend/resume support, it can enable autosuspend all by itself. For example, the video driver for a laptop's webcam might do this (in recent kernels they do), since these devices are rarely used and so should normally be autosuspended.

Sometimes it turns out that even when a device does work okay with autosuspend there are still problems. For example, the usbhid driver, which manages keyboards and mice, has autosuspend support. Tests with a number of keyboards show that typing on a suspended keyboard, while causing the keyboard to do a remote wakeup all right, will nonetheless frequently result in lost keystrokes. Tests with mice show that some of them will issue a remote-wakeup request in response to button presses but not to motion, and some in response to neither.

The kernel will not prevent you from enabling autosuspend on devices that can't handle it. It is even possible in theory to damage a device by suspending it at the wrong time. (Highly unlikely, but possible.) Take care.

20.8.8 The driver interface for Power Management

The requirements for a USB driver to support external power management are pretty modest; the driver need only define:

```
.suspend
.resume
.reset_resume
```

methods in its `usb_driver` structure, and the `reset_resume` method is optional. The methods' jobs are quite simple:

- The `suspend` method is called to warn the driver that the device is going to be suspended. If the driver returns a negative error code, the suspend will be aborted. Normally the driver will return 0, in which case it must cancel all outstanding URBs (`usb_kill_urb()`) and not submit any more.
- The `resume` method is called to tell the driver that the device has been resumed and the driver can return to normal operation. URBs may once more be submitted.
- The `reset_resume` method is called to tell the driver that the device has been resumed and it also has been reset. The driver should redo any necessary device initialization, since the device has probably lost most or all of its state (although the interfaces will be in the same altsettings as before the suspend).

If the device is disconnected or powered down while it is suspended, the `disconnect` method will be called instead of the `resume` or `reset_resume` method.

This is also quite likely to happen when waking up from hibernation, as many systems do not maintain suspend current to the USB host controllers during hibernation. (It's possible to work around the hibernation-forces-disconnect problem by using the USB Persist facility.)

The `reset_resume` method is used by the USB Persist facility (see USB device persistence during system suspend) and it can also be used under certain circumstances when `CONFIG_USB_PERSIST` is not enabled. Currently, if a device is reset during a resume and the driver does not have a `reset_resume` method, the driver won't receive any notification about the resume. Later kernels will call the driver's `disconnect` method; 2.6.23 doesn't do this.

USB drivers are bound to interfaces, so their `suspend` and `resume` methods get called when the interfaces are suspended or resumed. In principle one might want to suspend some interfaces on a device (i.e., force the drivers for those interface to stop all activity) without suspending the other interfaces. The USB core doesn't allow this; all interfaces are suspended when the device itself is suspended and all interfaces are resumed when the device is resumed. It isn't possible to suspend or resume some but not all of a device's interfaces. The closest you can come is to unbind the interfaces' drivers.

20.8.9 The driver interface for autosuspend and autoresume

To support autosuspend and autoresume, a driver should implement all three of the methods listed above. In addition, a driver indicates that it supports autosuspend by setting the `.supports_autosuspend` flag in its usb_driver structure. It is then responsible for informing the USB core whenever one of its interfaces becomes busy or idle. The driver does so by calling these six functions:

```
int  usb_autopm_get_interface(struct usb_interface *intf);
void usb_autopm_put_interface(struct usb_interface *intf);
int  usb_autopm_get_interface_async(struct usb_interface *intf);
void usb_autopm_put_interface_async(struct usb_interface *intf);
void usb_autopm_get_interface_no_resume(struct usb_interface *intf);
void usb_autopm_put_interface_no_suspend(struct usb_interface *intf);
```

The functions work by maintaining a usage counter in the usb_interface's embedded device structure. When the counter is > 0 then the interface is deemed to be busy, and the kernel will not autosuspend the interface's device. When the usage counter is = 0 then the interface is considered to be idle, and the kernel may autosuspend the device.

Drivers must be careful to balance their overall changes to the usage counter. Unbalanced "get"s will remain in effect when a driver is unbound from its interface, preventing the device from going into runtime suspend should the interface be bound to a driver again. On the other hand, drivers are allowed to achieve this balance by calling the `usb_autopm_*` functions even after their `disconnect` routine has returned – say from within a work-queue routine – provided they retain an active reference to the interface (via `usb_get_intf` and `usb_put_intf`).

Drivers using the async routines are responsible for their own synchronization and mutual exclusion.

`usb_autopm_get_interface()` increments the usage counter and does

an autoresume if the device is suspended. If the autoresume fails, the counter is decremented back.

`usb_autopm_put_interface()` decrements the usage counter and attempts an autosuspend if the new value is = 0.

`usb_autopm_get_interface_async()` and `usb_autopm_put_interface_async()` do almost the same things as their non-async counterparts. The big difference is that they use a workqueue to do the resume or suspend part of their jobs. As a result they can be called in an atomic context, such as an URB's completion handler, but when they return the device will generally not yet be in the desired state.

`usb_autopm_get_interface_no_resume()` and `usb_autopm_put_interface_no_suspend()` merely increment or decrement the usage counter; they do not attempt to carry out an autoresume or an autosuspend. Hence they can be called in an atomic context.

The simplest usage pattern is that a driver calls `usb_autopm_get_interface()` in its open routine and `usb_autopm_put_interface()` in its close or release routine. But other patterns are possible.

The autosuspend attempts mentioned above will often fail for one reason or another. For example, the `power/control` attribute might be set to `on`, or another interface in the same device might not be idle. This is perfectly normal. If the reason for failure was that the device hasn't been idle for long enough, a timer is scheduled to carry out the operation automatically when the autosuspend idle-delay has expired.

Autoresume attempts also can fail, although failure would mean that the device is no longer present or operating properly. Unlike autosuspend, there's no idle-delay for an autoresume.

20.8.10 Other parts of the driver interface

Drivers can enable autosuspend for their devices by calling:

```
usb_enable_autosuspend(struct usb_device *udev);
```

in their `probe()` routine, if they know that the device is capable of suspending and resuming correctly. This is exactly equivalent to writing `auto` to the device's `power/control` attribute. Likewise, drivers can disable autosuspend by calling:

```
usb_disable_autosuspend(struct usb_device *udev);
```

This is exactly the same as writing `on` to the `power/control` attribute.

Sometimes a driver needs to make sure that remote wakeup is enabled during autosuspend. For example, there's not much point autosuspending a keyboard if the user can't cause the keyboard to do a remote wakeup by typing on it. If the driver sets `intf->needs_remote_wakeup` to 1, the kernel won't autosuspend the device if remote wakeup isn't available. (If the device is already autosuspended, though, setting this flag won't cause the kernel to autoresume it. Normally a driver

would set this flag in its `probe` method, at which time the device is guaranteed not to be autosuspended.)

If a driver does its I/O asynchronously in interrupt context, it should call `usb_autopm_get_interface_async()` before starting output and `usb_autopm_put_interface_async()` when the output queue drains. When it receives an input event, it should call:

```
usb_mark_last_busy(struct usb_device *udev);
```

in the event handler. This tells the PM core that the device was just busy and therefore the next autosuspend idle-delay expiration should be pushed back. Many of the usb_autopm_* routines also make this call, so drivers need to worry only when interrupt-driven input arrives.

Asynchronous operation is always subject to races. For example, a driver may call the `usb_autopm_get_interface_async()` routine at a time when the core has just finished deciding the device has been idle for long enough but not yet gotten around to calling the driver's `suspend` method. The `suspend` method must be responsible for synchronizing with the I/O request routine and the URB completion handler; it should cause autosuspends to fail with -EBUSY if the driver needs to use the device.

External suspend calls should never be allowed to fail in this way, only autosuspend calls. The driver can tell them apart by applying the `PMSG_IS_AUTO()` macro to the message argument to the `suspend` method; it will return True for internal PM events (autosuspend) and False for external PM events.

20.8.11 Mutual exclusion

For external events – but not necessarily for autosuspend or autoresume – the device semaphore (udev->dev.sem) will be held when a `suspend` or `resume` method is called. This implies that external suspend/resume events are mutually exclusive with calls to `probe`, `disconnect`, `pre_reset`, and `post_reset`; the USB core guarantees that this is true of autosuspend/autoresume events as well.

If a driver wants to block all suspend/resume calls during some critical section, the best way is to lock the device and call `usb_autopm_get_interface()` (and do the reverse at the end of the critical section). Holding the device semaphore will block all external PM calls, and the `usb_autopm_get_interface()` will prevent any internal PM calls, even if it fails. (Exercise: Why?)

20.8.12 Interaction between dynamic PM and system PM

Dynamic power management and system power management can interact in a couple of ways.

Firstly, a device may already be autosuspended when a system suspend occurs. Since system suspends are supposed to be as transparent as possible, the device should remain suspended following the system resume. But this theory may not work out well in practice; over time the kernel's behavior in this regard has changed. As of 2.6.37 the policy is to resume all devices during a system resume and let them handle their own runtime suspends afterward.

Secondly, a dynamic power-management event may occur as a system suspend is underway. The window for this is short, since system suspends don't take long (a few seconds usually), but it can happen. For example, a suspended device may send a remote-wakeup signal while the system is suspending. The remote wakeup may succeed, which would cause the system suspend to abort. If the remote wakeup doesn't succeed, it may still remain active and thus cause the system to resume as soon as the system suspend is complete. Or the remote wakeup may fail and get lost. Which outcome occurs depends on timing and on the hardware and firmware design.

20.8.13 xHCI hardware link PM

xHCI host controller provides hardware link power management to usb2.0 (xHCI 1.0 feature) and usb3.0 devices which support link PM. By enabling hardware LPM, the host can automatically put the device into lower power state(L1 for usb2.0 devices, or U1/U2 for usb3.0 devices), which state device can enter and resume very quickly.

The user interface for controlling hardware LPM is located in the `power/` subdirectory of each USB device's sysfs directory, that is, in `/sys/bus/usb/devices/.../power/` where "…" is the device's ID. The relevant attribute files are `usb2_hardware_lpm` and `usb3_hardware_lpm`.

`power/usb2_hardware_lpm`

When a USB2 device which support LPM is plugged to a xHCI host root hub which support software LPM, the host will run a software LPM test for it; if the device enters L1 state and resume successfully and the host supports USB2 hardware LPM, this file will show up and driver will enable hardware LPM for the device. You can write y/Y/1 or n/N/0 to the file to enable/disable USB2 hardware LPM manually. This is for test purpose mainly.

`power/usb3_hardware_lpm_u1` `power/usb3_hardware_lpm_u2`

When a USB 3.0 lpm-capable device is plugged in to a xHCI host which supports link PM, it will check if U1 and U2 exit latencies have been set in the BOS descriptor; if the check is passed and the host supports USB3 hardware LPM, USB3 hardware LPM will be enabled for the device and these files will be created. The files hold a string value (enable or disable) indicating whether or not USB3 hardware LPM U1 or U2 is enabled for the device.

20.8.14 USB Port Power Control

In addition to suspending endpoint devices and enabling hardware controlled link power management, the USB subsystem also has the capability to disable power to ports under some conditions. Power is controlled through `Set/ClearPortFeature(PORT_POWER)` requests to a hub. In the case of a root or platform-internal hub the host controller driver translates `PORT_POWER` requests into platform firmware (ACPI) method calls to set the port power state. For more background see the Linux Plumbers Conference 2012 slides[1] and video[2]:

Upon receiving a `ClearPortFeature(PORT_POWER)` request a USB port is logically off, and may trigger the actual loss of VBUS to the port[3]. VBUS may be maintained in the case where a hub gangs multiple ports into a shared power well causing power to remain until all ports in the gang are turned off. VBUS may also be maintained by hub ports configured for a charging application. In any event a logically off port will lose connection with its device, not respond to hotplug events, and not respond to remote wakeup events.

Warning: turning off a port may result in the inability to hot add a device. Please see "User Interface for Port Power Control" for details.

As far as the effect on the device itself it is similar to what a device goes through during system suspend, i.e. the power session is lost. Any USB device or driver that misbehaves with system suspend will be similarly affected by a port power cycle event. For this reason the implementation shares the same device recovery path (and honors the same quirks) as the system resume path for the hub.

20.8.15 User Interface for Port Power Control

The port power control mechanism uses the PM runtime system. Poweroff is requested by clearing the `power/pm_qos_no_power_off` flag of the port device (defaults to 1). If the port is disconnected it will immediately receive a `ClearPortFeature(PORT_POWER)` request. Otherwise, it will honor the pm runtime rules and require the attached child device and all descendants to be suspended. This mechanism is dependent on the hub advertising port power switching in its hub descriptor (wHubCharacteristics logical power switching mode field).

Note, some interface devices/drivers do not support autosuspend. Userspace may need to unbind the interface drivers before the `usb_device` will suspend. An unbound interface device is suspended by default. When unbinding, be careful to unbind interface drivers, not the driver of the parent usb device. Also, leave hub interface drivers bound. If the driver for the usb device (not interface) is unbound the kernel is no longer able to resume the device. If a hub interface driver is unbound, control of its child ports is lost and all attached child-devices will disconnect. A good rule of thumb is that if the 'driver/module' link for a device

[1] http://dl.dropbox.com/u/96820575/sarah-sharp-lpt-port-power-off2-mini.pdf

[2] http://linuxplumbers.ubicast.tv/videos/usb-port-power-off-kerneluserspace-api/

[3] USB 3.1 Section 10.12

wakeup note: if a device is configured to send wakeup events the port power control implementation will block poweroff attempts on that port.

points to /sys/module/usbcore then unbinding it will interfere with port power control.

Example of the relevant files for port power control. Note, in this example these files are relative to a usb hub device (prefix):

```
prefix=/sys/devices/pci0000:00/0000:00:14.0/usb3/3-1

                attached child device +
             hub port device +         |
hub interface device +       |         |
                     v       v         v
             $prefix/3-1:1.0/3-1-port1/device

$prefix/3-1:1.0/3-1-port1/power/pm_qos_no_power_off
$prefix/3-1:1.0/3-1-port1/device/power/control
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intf0>/driver/unbind
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intf1>/driver/unbind
...
$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intfN>/driver/unbind
```

In addition to these files some ports may have a 'peer' link to a port on another hub. The expectation is that all superspeed ports have a hi-speed peer:

```
$prefix/3-1:1.0/3-1-port1/peer -> ../../../../usb2/2-1/2-1:1.0/2-1-port1
../../../../usb2/2-1/2-1:1.0/2-1-port1/peer -> ../../../../usb3/3-1/3-1:1.
↪0/3-1-port1
```

Distinct from 'companion ports', or 'ehci/xhci shared switchover ports' peer ports are simply the hi-speed and superspeed interface pins that are combined into a single usb3 connector. Peer ports share the same ancestor XHCI device.

While a superspeed port is powered off a device may downgrade its connection and attempt to connect to the hi-speed pins. The implementation takes steps to prevent this:

1. Port suspend is sequenced to guarantee that hi-speed ports are powered-off before their superspeed peer is permitted to power-off. The implication is that the setting pm_qos_no_power_off to zero on a superspeed port may not cause the port to power-off until its highspeed peer has gone to its runtime suspend state. Userspace must take care to order the suspensions if it wants to guarantee that a superspeed port will power-off.

2. Port resume is sequenced to force a superspeed port to power-on prior to its highspeed peer.

3. Port resume always triggers an attached child device to resume. After a power session is lost the device may have been removed, or need reset. Resuming the child device when the parent port regains power resolves those states and clamps the maximum port power cycle frequency at the rate the child device can suspend (autosuspend-delay) and resume (reset-resume latency).

Sysfs files relevant for port power control:

<hubdev-portX>/power/pm_qos_no_power_off: This writable flag controls the state of an idle port. Once all children and descen-

dants have suspended the port may suspend/poweroff provided that pm_qos_no_power_off is ‘0’. If pm_qos_no_power_off is ‘1’ the port will remain active/powered regardless of the stats of descendants. Defaults to 1.

<hubdev-portX>/power/runtime_status: This file reflects whether the port is ‘active’ (power is on) or ‘suspended’ (logically off). There is no indication to userspace whether VBUS is still supplied.

<hubdev-portX>/connect_type: An advisory read-only flag to userspace indicating the location and connection type of the port. It returns one of four values ‘hotplug’, ‘hardwired’, ‘not used’, and ‘unknown’. All values, besides unknown, are set by platform firmware.

`hotplug` indicates an externally connectable/visible port on the platform. Typically userspace would choose to keep such a port powered to handle new device connection events.

`hardwired` refers to a port that is not visible but connectable. Examples are internal ports for USB bluetooth that can be disconnected via an external switch or a port with a hardwired USB camera. It is expected to be safe to allow these ports to suspend provided pm_qos_no_power_off is coordinated with any switch that gates connections. Userspace must arrange for the device to be connected prior to the port powering off, or to activate the port prior to enabling connection via a switch.

`not used` refers to an internal port that is expected to never have a device connected to it. These may be empty internal ports, or ports that are not physically exposed on a platform. Considered safe to be powered-off at all times.

`unknown` means platform firmware does not provide information for this port. Most commonly refers to external hub ports which should be considered ‘hotplug’ for policy decisions.

Note:

- since we are relying on the BIOS to get this ACPI information correct, the USB port descriptions may be missing or wrong.
- Take care in clearing `pm_qos_no_power_off`. Once power is off this port will not respond to new connect events.

Once a child device is attached additional constraints are applied before the port is allowed to poweroff.

<child>/power/control: Must be `auto`, and the port will not power down until `<child>/power/runtime_status` reflects the ‘suspended’ state. Default value is controlled by child device driver.

<child>/power/persist: This defaults to 1 for most devices and indicates if kernel can persist the device’s configuration across a power

session loss (suspend / port-power event). When this value is `0` (quirky devices), port poweroff is disabled.

<child>/driver/unbind: Wakeup capable devices will block port poweroff. At this time the only mechanism to clear the usb-internal wakeup-capability for an interface device is to unbind its driver.

Summary of poweroff pre-requisite settings relative to a port device:

```
echo 0 > power/pm_qos_no_power_off
echo 0 > peer/power/pm_qos_no_power_off # if it exists
echo auto > power/control # this is the default value
echo auto > <child>/power/control
echo 1 > <child>/power/persist # this is the default value
```

20.8.16 Suggested Userspace Port Power Policy

As noted above userspace needs to be careful and deliberate about what ports are enabled for poweroff.

The default configuration is that all ports start with `power/pm_qos_no_power_off` set to `1` causing ports to always remain active.

Given confidence in the platform firmware's description of the ports (ACPI _PLD record for a port populates 'connect_type') userspace can clear pm_qos_no_power_off for all 'not used' ports. The same can be done for 'hard-wired' ports provided poweroff is coordinated with any connection switch for the port.

A more aggressive userspace policy is to enable USB port power off for all ports (set `<hubdev-portX>/power/pm_qos_no_power_off` to `0`) when some external factor indicates the user has stopped interacting with the system. For example, a distro may want to enable power off all USB ports when the screen blanks, and re-power them when the screen becomes active. Smart phones and tablets may want to power off USB ports when the user pushes the power button.

20.9 USB hotplugging

20.9.1 Linux Hotplugging

In hotpluggable busses like USB (and Cardbus PCI), end-users plug devices into the bus with power on. In most cases, users expect the devices to become immediately usable. That means the system must do many things, including:

- Find a driver that can handle the device. That may involve loading a kernel module; newer drivers can use module-init-tools to publish their device (and class) support to user utilities.
- Bind a driver to that device. Bus frameworks do that using a device driver's probe() routine.

- Tell other subsystems to configure the new device. Print queues may need to be enabled, networks brought up, disk partitions mounted, and so on. In some cases these will be driver-specific actions.

This involves a mix of kernel mode and user mode actions. Making devices be immediately usable means that any user mode actions can't wait for an administrator to do them: the kernel must trigger them, either passively (triggering some monitoring daemon to invoke a helper program) or actively (calling such a user mode helper program directly).

Those triggered actions must support a system's administrative policies; such programs are called "policy agents" here. Typically they involve shell scripts that dispatch to more familiar administration tools.

Because some of those actions rely on information about drivers (metadata) that is currently available only when the drivers are dynamically linked, you get the best hotplugging when you configure a highly modular system.

20.9.2 Kernel Hotplug Helper (`/sbin/hotplug`)

There is a kernel parameter: `/proc/sys/kernel/hotplug`, which normally holds the pathname `/sbin/hotplug`. That parameter names a program which the kernel may invoke at various times.

The /sbin/hotplug program can be invoked by any subsystem as part of its reaction to a configuration change, from a thread in that subsystem. Only one parameter is required: the name of a subsystem being notified of some kernel event. That name is used as the first key for further event dispatch; any other argument and environment parameters are specified by the subsystem making that invocation.

Hotplug software and other resources is available at:

http://linux-hotplug.sourceforge.net

Mailing list information is also available at that site.

20.9.3 USB Policy Agent

The USB subsystem currently invokes `/sbin/hotplug` when USB devices are added or removed from system. The invocation is done by the kernel hub workqueue [hub_wq], or else as part of root hub initialization (done by init, modprobe, kapmd, etc). Its single command line parameter is the string "usb", and it passes these environment variables:

ACTION	`add, remove`
PRODUCT	USB vendor, product, and version codes (hex)
TYPE	device class codes (decimal)
INTERFACE	interface 0 class codes (decimal)

If "usbdevfs" is configured, DEVICE and DEVFS are also passed. DEVICE is the pathname of the device, and is useful for devices with multiple and/or alternate interfaces that complicate driver selection. By design, USB hotplugging is independent of `usbdevfs`: you can do most essential parts of USB device setup without

using that filesystem, and without running a user mode daemon to detect changes in system configuration.

Currently available policy agent implementations can load drivers for modules, and can invoke driver-specific setup scripts. The newest ones leverage USB module-init-tools support. Later agents might unload drivers.

20.9.4 USB Modutils Support

Current versions of module-init-tools will create a `modules.usbmap` file which contains the entries from each driver's `MODULE_DEVICE_TABLE`. Such files can be used by various user mode policy agents to make sure all the right driver modules get loaded, either at boot time or later.

See `linux/usb.h` for full information about such table entries; or look at existing drivers. Each table entry describes one or more criteria to be used when matching a driver to a device or class of devices. The specific criteria are identified by bits set in "match_flags", paired with field values. You can construct the criteria directly, or with macros such as these, and use driver_info to store more information:

```
USB_DEVICE (vendorId, productId)
    ... matching devices with specified vendor and product ids
USB_DEVICE_VER (vendorId, productId, lo, hi)
    ... like USB_DEVICE with lo <= productversion <= hi
USB_INTERFACE_INFO (class, subclass, protocol)
    ... matching specified interface class info
USB_DEVICE_INFO (class, subclass, protocol)
    ... matching specified device class info
```

A short example, for a driver that supports several specific USB devices and their quirks, might have a MODULE_DEVICE_TABLE like this:

```
static const struct usb_device_id mydriver_id_table[] = {
    { USB_DEVICE (0x9999, 0xaaaa), driver_info: QUIRK_X },
    { USB_DEVICE (0xbbbb, 0x8888), driver_info: QUIRK_Y|QUIRK_Z },
    ...
    { } /* end with an all-zeroes entry */
};
MODULE_DEVICE_TABLE(usb, mydriver_id_table);
```

Most USB device drivers should pass these tables to the USB subsystem as well as to the module management subsystem. Not all, though: some driver frameworks connect using interfaces layered over USB, and so they won't need such a struct `usb_driver`.

Drivers that connect directly to the USB subsystem should be declared something like this:

```
static struct usb_driver mydriver = {
    .name           = "mydriver",
    .id_table       = mydriver_id_table,
    .probe          = my_probe,
    .disconnect     = my_disconnect,
```

(continues on next page)

(continued from previous page)

```
    /*
    if using the usb chardev framework:
        .minor              = MY_USB_MINOR_START,
        .fops               = my_file_ops,
    if exposing any operations through usbdevfs:
        .ioctl              = my_ioctl,
    */
};
```

When the USB subsystem knows about a driver's device ID table, it's used when choosing drivers to probe(). The thread doing new device processing checks drivers' device ID entries from the `MODULE_DEVICE_TABLE` against interface and device descriptors for the device. It will only call `probe()` if there is a match, and the third argument to `probe()` will be the entry that matched.

If you don't provide an `id_table` for your driver, then your driver may get probed for each new device; the third parameter to `probe()` will be `NULL`.

20.10 USB device persistence during system suspend

Author Alan Stern <stern@rowland.harvard.edu>

Date September 2, 2006 (Updated February 25, 2008)

20.10.1 What is the problem?

According to the USB specification, when a USB bus is suspended the bus must continue to supply suspend current (around 1-5 mA). This is so that devices can maintain their internal state and hubs can detect connect-change events (devices being plugged in or unplugged). The technical term is "power session".

If a USB device's power session is interrupted then the system is required to behave as though the device has been unplugged. It's a conservative approach; in the absence of suspend current the computer has no way to know what has actually happened. Perhaps the same device is still attached or perhaps it was removed and a different device plugged into the port. The system must assume the worst.

By default, Linux behaves according to the spec. If a USB host controller loses power during a system suspend, then when the system wakes up all the devices attached to that controller are treated as though they had disconnected. This is always safe and it is the "officially correct" thing to do.

For many sorts of devices this behavior doesn't matter in the least. If the kernel wants to believe that your USB keyboard was unplugged while the system was asleep and a new keyboard was plugged in when the system woke up, who cares? It'll still work the same when you type on it.

Unfortunately problems _can_ arise, particularly with mass-storage devices. The effect is exactly the same as if the device really had been unplugged while the system was suspended. If you had a mounted filesystem on the device, you're out of luck – everything in that filesystem is now inaccessible. This is especially

annoying if your root filesystem was located on the device, since your system will instantly crash.

Loss of power isn't the only mechanism to worry about. Anything that interrupts a power session will have the same effect. For example, even though suspend current may have been maintained while the system was asleep, on many systems during the initial stages of wakeup the firmware (i.e., the BIOS) resets the motherboard's USB host controllers. Result: all the power sessions are destroyed and again it's as though you had unplugged all the USB devices. Yes, it's entirely the BIOS's fault, but that doesn't do _you_ any good unless you can convince the BIOS supplier to fix the problem (lots of luck!).

On many systems the USB host controllers will get reset after a suspend-to-RAM. On almost all systems, no suspend current is available during hibernation (also known as swsusp or suspend-to-disk). You can check the kernel log after resuming to see if either of these has happened; look for lines saying "root hub lost power or was reset".

In practice, people are forced to unmount any filesystems on a USB device before suspending. If the root filesystem is on a USB device, the system can't be suspended at all. (All right, it _can_ be suspended – but it will crash as soon as it wakes up, which isn't much better.)

20.10.2 What is the solution?

The kernel includes a feature called USB-persist. It tries to work around these issues by allowing the core USB device data structures to persist across a power-session disruption.

It works like this. If the kernel sees that a USB host controller is not in the expected state during resume (i.e., if the controller was reset or otherwise had lost power) then it applies a persistence check to each of the USB devices below that controller for which the "persist" attribute is set. It doesn't try to resume the device; that can't work once the power session is gone. Instead it issues a USB port reset and then re-enumerates the device. (This is exactly the same thing that happens whenever a USB device is reset.) If the re-enumeration shows that the device now attached to that port has the same descriptors as before, including the Vendor and Product IDs, then the kernel continues to use the same device structure. In effect, the kernel treats the device as though it had merely been reset instead of unplugged.

The same thing happens if the host controller is in the expected state but a USB device was unplugged and then replugged, or if a USB device fails to carry out a normal resume.

If no device is now attached to the port, or if the descriptors are different from what the kernel remembers, then the treatment is what you would expect. The kernel destroys the old device structure and behaves as though the old device had been unplugged and a new device plugged in.

The end result is that the USB device remains available and usable. Filesystem mounts and memory mappings are unaffected, and the world is now a good and happy place.

Note that the "USB-persist" feature will be applied only to those devices for which it is enabled. You can enable the feature by doing (as root):

```
echo 1 >/sys/bus/usb/devices/.../power/persist
```

where the "…" should be filled in the with the device's ID. Disable the feature by writing 0 instead of 1. For hubs the feature is automatically and permanently enabled and the power/persist file doesn't even exist, so you only have to worry about setting it for devices where it really matters.

20.10.3 Is this the best solution?

Perhaps not. Arguably, keeping track of mounted filesystems and memory mappings across device disconnects should be handled by a centralized Logical Volume Manager. Such a solution would allow you to plug in a USB flash device, create a persistent volume associated with it, unplug the flash device, plug it back in later, and still have the same persistent volume associated with the device. As such it would be more far-reaching than USB-persist.

On the other hand, writing a persistent volume manager would be a big job and using it would require significant input from the user. This solution is much quicker and easier – and it exists now, a giant point in its favor!

Furthermore, the USB-persist feature applies to _all_ USB devices, not just mass-storage devices. It might turn out to be equally useful for other device types, such as network interfaces.

20.10.4 WARNING: USB-persist can be dangerous!!

When recovering an interrupted power session the kernel does its best to make sure the USB device hasn't been changed; that is, the same device is still plugged into the port as before. But the checks aren't guaranteed to be 100% accurate.

If you replace one USB device with another of the same type (same manufacturer, same IDs, and so on) there's an excellent chance the kernel won't detect the change. The serial number string and other descriptors are compared with the kernel's stored values, but this might not help since manufacturers frequently omit serial numbers entirely in their devices.

Furthermore it's quite possible to leave a USB device exactly the same while changing its media. If you replace the flash memory card in a USB card reader while the system is asleep, the kernel will have no way to know you did it. The kernel will assume that nothing has happened and will continue to use the partition tables, inodes, and memory mappings for the old card.

If the kernel gets fooled in this way, it's almost certain to cause data corruption and to crash your system. You'll have no one to blame but yourself.

For those devices with avoid_reset_quirk attribute being set, persist maybe fail because they may morph after reset.

YOU HAVE BEEN WARNED! USE AT YOUR OWN RISK!

That having been said, most of the time there shouldn't be any trouble at all. The USB-persist feature can be extremely useful. Make the most of it.

20.11 USB Error codes

Revised 2004-Oct-21

This is the documentation of (hopefully) all possible error codes (and their interpretation) that can be returned from usbcore.

Some of them are returned by the Host Controller Drivers (HCDs), which device drivers only see through usbcore. As a rule, all the HCDs should behave the same except for transfer speed dependent behaviors and the way certain faults are reported.

20.11.1 Error codes returned by `usb_submit_urb()`

Non-USB-specific:

0	URB submission went fine
`-ENOMEM`	no memory for allocation of internal structures

USB-specific:

-EBUSY	The URB is already active.
-ENODEV	specified USB-device or bus doesn't exist
-ENOENT	specified interface or endpoint does not exist or is not enabled
-ENXIO	host controller driver does not support queuing of this type of urb. (treat as a host controller bug.)
-EINVAL	a) Invalid transfer type specified (or not supported) b) Invalid or unsupported periodic transfer interval c) ISO: attempted to change transfer interval d) ISO: number_of_packets is < 0 e) various other cases
-EXDEV	ISO: URB_ISO_ASAP wasn't specified and all the frames the URB would be scheduled in have already expired.
-EFBIG	Host controller driver can't schedule that many ISO frames.
-EPIPE	The pipe type specified in the URB doesn't match the endpoint's actual type.
-EMSGSIZE	(a) endpoint maxpacket size is zero; it is not usable in the current interface altsetting. (b) ISO packet is larger than the endpoint maxpacket. (c) requested data transfer length is invalid: negative or too large for the host controller.
-ENOSPC	This request would overcommit the usb bandwidth reserved for periodic transfers (interrupt, isochronous).
-ESHUTDOWN	The device or host controller has been disabled due to some problem that could not be worked around.
-EPERM	Submission failed because urb->reject was set.
-EHOSTUNREACH	URB was rejected because the device is suspended.
-ENOEXEC	A control URB doesn't contain a Setup packet.

20.11.2 Error codes returned by `in urb->status` or in `iso_frame_desc[n].status` (for ISO)

USB device drivers may only test urb status values in completion handlers. This is because otherwise there would be a race between HCDs updating these values on one CPU, and device drivers testing them on another CPU.

A transfer' s actual_length may be positive even when an error has been reported. That's because transfers often involve several packets, so that one or more packets could finish before an error stops further endpoint I/O.

For isochronous URBs, the urb status value is non-zero only if the URB is unlinked, the device is removed, the host controller is disabled, or the total transferred length is less than the requested length and the `URB_SHORT_NOT_OK` flag is set. Completion handlers for isochronous URBs should only see `urb->status` set to zero, `-ENOENT`, `-ECONNRESET`, `-ESHUTDOWN`, or `-EREMOTEIO`. Individual frame descriptor status fields may report more status codes.

0	Transfer completed successfully
-ENOENT	URB was synchronously unlinked by usb_unlink_urb()
-EINPROGRESS	URB still pending, no results yet (That is, if drivers see this it's a bug.)
-EPROTO[1],[2]	a) bitstuff error b) no response packet received within the prescribed bus turn-around time c) unknown USB error
-EILSEQ[1],[2]	a) CRC mismatch b) no response packet received within the prescribed bus turn-around time c) unknown USB error Note that often the controller hardware does not distinguish among cases a), b), and c), so a driver cannot tell whether there was a protocol error, a failure to respond (often caused by device disconnect), or some other fault.
-ETIME[2]	No response packet received within the prescribed bus turn-around time. This error may instead be reported as -EPROTO or -EILSEQ.
-ETIMEDOUT	Synchronous USB message functions use this code to indicate timeout expired before the transfer completed, and no other error was reported by HC.
-EPIPE[2]	Endpoint stalled. For non-control endpoints, reset this status with usb_clear_halt().
-ECOMM	During an IN transfer, the host controller received data from an endpoint faster than it could be written to system memory
-ENOSR	During an OUT transfer, the host controller could not retrieve data from system memory fast enough to keep up with the USB data rate
-EOVERFLOW[1]	The amount of data returned by the endpoint was greater than either the max packet size of the endpoint or the remaining buffer size. "Babble".
-EREMOTEIO	The data read from the endpoint did not fill the specified buffer, and URB_SHORT_NOT_OK was set in urb->transfer_flags.
-ENODEV	Device was removed. Often preceded by a burst of other errors, since the hub driver doesn't detect device removal events immediately.
-EXDEV	ISO transfer only partially completed

20.11.3 Error codes returned by usbcore-functions

Note: expect also other submit and transfer status codes

`usb_register()`:

`-EINVAL`	error during registering new driver

`usb_get_*/usb_set_*()`, `usb_control_msg()`, `usb_bulk_msg()`:

`-ETIMEDOUT`	Timeout expired before the transfer completed.

20.12 Writing USB Device Drivers

Author Greg Kroah-Hartman

20.12.1 Introduction

The Linux USB subsystem has grown from supporting only two different types of devices in the 2.2.7 kernel (mice and keyboards), to over 20 different types of devices in the 2.4 kernel. Linux currently supports almost all USB class devices (standard types of devices like keyboards, mice, modems, printers and speakers) and an ever-growing number of vendor-specific devices (such as USB to serial converters, digital cameras, Ethernet devices and MP3 players). For a full list of the different USB devices currently supported, see Resources.

The remaining kinds of USB devices that do not have support on Linux are almost all vendor-specific devices. Each vendor decides to implement a custom protocol to talk to their device, so a custom driver usually needs to be created. Some vendors are open with their USB protocols and help with the creation of Linux drivers, while others do not publish them, and developers are forced to reverse-engineer. See Resources for some links to handy reverse-engineering tools.

Because each different protocol causes a new driver to be created, I have written a generic USB driver skeleton, modelled after the pci-skeleton.c file in the kernel source tree upon which many PCI network drivers have been based. This USB skeleton can be found at drivers/usb/usb-skeleton.c in the kernel source tree. In this article I will walk through the basics of the skeleton driver, explaining the different pieces and what needs to be done to customize it to your specific device.

[1] Error codes like `-EPROTO`, `-EILSEQ` and `-EOVERFLOW` normally indicate hardware problems such as bad devices (including firmware) or cables.

[2] This is also one of several codes that different kinds of host controller use to indicate a transfer has failed because of device disconnect. In the interval before the hub driver starts disconnect processing, devices may receive such fault reports for every request.

20.12.2 Linux USB Basics

If you are going to write a Linux USB driver, please become familiar with the USB protocol specification. It can be found, along with many other useful documents, at the USB home page (see Resources). An excellent introduction to the Linux USB subsystem can be found at the USB Working Devices List (see Resources). It explains how the Linux USB subsystem is structured and introduces the reader to the concept of USB urbs (USB Request Blocks), which are essential to USB drivers.

The first thing a Linux USB driver needs to do is register itself with the Linux USB subsystem, giving it some information about which devices the driver supports and which functions to call when a device supported by the driver is inserted or removed from the system. All of this information is passed to the USB subsystem in the `usb_driver` structure. The skeleton driver declares a `usb_driver` as:

```
static struct usb_driver skel_driver = {
        .name        = "skeleton",
        .probe       = skel_probe,
        .disconnect  = skel_disconnect,
        .fops        = &skel_fops,
        .minor       = USB_SKEL_MINOR_BASE,
        .id_table    = skel_table,
};
```

The variable name is a string that describes the driver. It is used in informational messages printed to the system log. The probe and disconnect function pointers are called when a device that matches the information provided in the `id_table` variable is either seen or removed.

The fops and minor variables are optional. Most USB drivers hook into another kernel subsystem, such as the SCSI, network or TTY subsystem. These types of drivers register themselves with the other kernel subsystem, and any user-space interactions are provided through that interface. But for drivers that do not have a matching kernel subsystem, such as MP3 players or scanners, a method of interacting with user space is needed. The USB subsystem provides a way to register a minor device number and a set of `file_operations` function pointers that enable this user-space interaction. The skeleton driver needs this kind of interface, so it provides a minor starting number and a pointer to its `file_operations` functions.

The USB driver is then registered with a call to `usb_register()`, usually in the driver's init function, as shown here:

```
static int __init usb_skel_init(void)
{
        int result;

        /* register this driver with the USB subsystem */
        result = usb_register(&skel_driver);
        if (result < 0) {
                err("usb_register failed for the "__FILE__ "driver."
                    "Error number %d", result);
                return -1;
        }

        return 0;
```

(continues on next page)

(continued from previous page)

```
}
module_init(usb_skel_init);
```

When the driver is unloaded from the system, it needs to deregister itself with the USB subsystem. This is done with the `usb_deregister()` function:

```
static void __exit usb_skel_exit(void)
{
        /* deregister this driver with the USB subsystem */
        usb_deregister(&skel_driver);
}
module_exit(usb_skel_exit);
```

To enable the linux-hotplug system to load the driver automatically when the device is plugged in, you need to create a `MODULE_DEVICE_TABLE`. The following code tells the hotplug scripts that this module supports a single device with a specific vendor and product ID:

```
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
        { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
        { }                     /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

There are other macros that can be used in describing a struct `usb_device_id` for drivers that support a whole class of USB drivers. See usb.h for more information on this.

20.12.3 Device operation

When a device is plugged into the USB bus that matches the device ID pattern that your driver registered with the USB core, the probe function is called. The `usb_device` structure, interface number and the interface ID are passed to the function:

```
static int skel_probe(struct usb_interface *interface,
    const struct usb_device_id *id)
```

The driver now needs to verify that this device is actually one that it can accept. If so, it returns 0. If not, or if any error occurs during initialization, an errorcode (such as `-ENOMEM` or `-ENODEV`) is returned from the probe function.

In the skeleton driver, we determine what end points are marked as bulk-in and bulk-out. We create buffers to hold the data that will be sent and received from the device, and a USB urb to write data to the device is initialized.

Conversely, when the device is removed from the USB bus, the disconnect function is called with the device pointer. The driver needs to clean any private data that has been allocated at this time and to shut down any pending urbs that are in the USB system.

Now that the device is plugged into the system and the driver is bound to the device, any of the functions in the `file_operations` structure that were passed to

the USB subsystem will be called from a user program trying to talk to the device. The first function called will be open, as the program tries to open the device for I/O. We increment our private usage count and save a pointer to our internal structure in the file structure. This is done so that future calls to file operations will enable the driver to determine which device the user is addressing. All of this is done with the following code:

```
/* increment our usage count for the module */
++skel->open_count;

/* save our object in the file's private structure */
file->private_data = dev;
```

After the open function is called, the read and write functions are called to receive and send data to the device. In the `skel_write` function, we receive a pointer to some data that the user wants to send to the device and the size of the data. The function determines how much data it can send to the device based on the size of the write urb it has created (this size depends on the size of the bulk out end point that the device has). Then it copies the data from user space to kernel space, points the urb to the data and submits the urb to the USB subsystem. This can be seen in the following code:

```
/* we can only write as much as 1 urb will hold */
bytes_written = (count > skel->bulk_out_size) ? skel->bulk_out_size :␣
↪count;

/* copy the data from user space into our urb */
copy_from_user(skel->write_urb->transfer_buffer, buffer, bytes_written);

/* set up our urb */
usb_fill_bulk_urb(skel->write_urb,
                  skel->dev,
                  usb_sndbulkpipe(skel->dev, skel->bulk_out_endpointAddr),
                  skel->write_urb->transfer_buffer,
                  bytes_written,
                  skel_write_bulk_callback,
                  skel);

/* send the data out the bulk port */
result = usb_submit_urb(skel->write_urb);
if (result) {
        err("Failed submitting write urb, error %d", result);
}
```

When the write urb is filled up with the proper information using the `usb_fill_bulk_urb()` function, we point the urb's completion callback to call our own `skel_write_bulk_callback` function. This function is called when the urb is finished by the USB subsystem. The callback function is called in interrupt context, so caution must be taken not to do very much processing at that time. Our implementation of `skel_write_bulk_callback` merely reports if the urb was completed successfully or not and then returns.

The read function works a bit differently from the write function in that we do not use an urb to transfer data from the device to the driver. Instead we call the `usb_bulk_msg()` function, which can be used to send or receive data from a

device without having to create urbs and handle urb completion callback functions. We call the usb_bulk_msg() function, giving it a buffer into which to place any data received from the device and a timeout value. If the timeout period expires without receiving any data from the device, the function will fail and return an error message. This can be shown with the following code:

```
/* do an immediate bulk read to get data from the device */
retval = usb_bulk_msg (skel->dev,
                       usb_rcvbulkpipe (skel->dev,
                       skel->bulk_in_endpointAddr),
                       skel->bulk_in_buffer,
                       skel->bulk_in_size,
                       &count, HZ*10);
/* if the read was successful, copy the data to user space */
if (!retval) {
        if (copy_to_user (buffer, skel->bulk_in_buffer, count))
                retval = -EFAULT;
        else
                retval = count;
}
```

The usb_bulk_msg() function can be very useful for doing single reads or writes to a device; however, if you need to read or write constantly to a device, it is recommended to set up your own urbs and submit them to the USB subsystem.

When the user program releases the file handle that it has been using to talk to the device, the release function in the driver is called. In this function we decrement our private usage count and wait for possible pending writes:

```
/* decrement our usage count for the device */
--skel->open_count;
```

One of the more difficult problems that USB drivers must be able to handle smoothly is the fact that the USB device may be removed from the system at any point in time, even if a program is currently talking to it. It needs to be able to shut down any current reads and writes and notify the user-space programs that the device is no longer there. The following code (function skel_delete) is an example of how to do this:

```
static inline void skel_delete (struct usb_skel *dev)
{
    kfree (dev->bulk_in_buffer);
    if (dev->bulk_out_buffer != NULL)
        usb_free_coherent (dev->udev, dev->bulk_out_size,
            dev->bulk_out_buffer,
            dev->write_urb->transfer_dma);
    usb_free_urb (dev->write_urb);
    kfree (dev);
}
```

If a program currently has an open handle to the device, we reset the flag device_present. For every read, write, release and other functions that expect a device to be present, the driver first checks this flag to see if the device is still present. If not, it releases that the device has disappeared, and a -ENODEV error is returned to the user-space program. When the release function is eventually called, it determines if there is no device and if not, it does the cleanup that the

`skel_disconnect` function normally does if there are no open files on the device (see Listing 5).

20.12.4 Isochronous Data

This usb-skeleton driver does not have any examples of interrupt or isochronous data being sent to or from the device. Interrupt data is sent almost exactly as bulk data is, with a few minor exceptions. Isochronous data works differently with continuous streams of data being sent to or from the device. The audio and video camera drivers are very good examples of drivers that handle isochronous data and will be useful if you also need to do this.

20.12.5 Conclusion

Writing Linux USB device drivers is not a difficult task as the usb-skeleton driver shows. This driver, combined with the other current USB drivers, should provide enough examples to help a beginning author create a working driver in a minimal amount of time. The linux-usb-devel mailing list archives also contain a lot of helpful information.

20.12.6 Resources

The Linux USB Project: http://www.linux-usb.org/

Linux Hotplug Project: http://linux-hotplug.sourceforge.net/

linux-usb Mailing List Archives: https://lore.kernel.org/linux-usb/

Programming Guide for Linux USB Device Drivers: http://lmu.web.psi.ch/docu/manuals/software_manuals/linux_sl/usb_linux_programming_guide.pdf

USB Home Page: http://www.usb.org

20.13 Synopsys DesignWare Core SuperSpeed USB 3.0 Controller

Author Felipe Balbi <felipe.balbi@linux.intel.com>

Date April 2017

20.13.1 Introduction

The Synopsys DesignWare Core SuperSpeed USB 3.0 Controller (hereinafter referred to as DWC3) is a USB SuperSpeed compliant controller which can be configured in one of 4 ways:

1. Peripheral-only configuration
2. Host-only configuration
3. Dual-Role configuration

4. Hub configuration

Linux currently supports several versions of this controller. In all likelyhood, the version in your SoC is already supported. At the time of this writing, known tested versions range from 2.02a to 3.10a. As a rule of thumb, anything above 2.02a should work reliably well.

Currently, we have many known users for this driver. In alphabetical order:

1. Cavium
2. Intel Corporation
3. Qualcomm
4. Rockchip
5. ST
6. Samsung
7. Texas Instruments
8. Xilinx

20.13.2 Summary of Features

For details about features supported by your version of DWC3, consult your IP team and/or Synopsys DesignWare Core SuperSpeed USB 3.0 Controller Databook. Following is a list of features supported by the driver at the time of this writing:

1. Up to 16 bidirectional endpoints (including the control pipe - ep0)
2. Flexible endpoint configuration
3. Simultaneous IN and OUT transfer support
4. Scatter-list support
5. Up to 256 TRBs[1] per endpoint
6. Support for all transfer types (Control, Bulk, Interrupt, and Isochronous)
7. SuperSpeed Bulk Streams
8. Link Power Management
9. Trace Events for debugging
10. DebugFS[3] interface

These features have all been exercised with many of the **in-tree** gadget drivers. We have verified both ConfigFS[4] and legacy gadget drivers.

[1] Transfer Request Block
[3] The Debug File System
[4] The Config File System

20.13.3 Driver Design

The DWC3 driver sits on the drivers/usb/dwc3/ directory. All files related to this driver are in this one directory. This makes it easy for new-comers to read the code and understand how it behaves.

Because of DWC3's configuration flexibility, the driver is a little complex in some places but it should be rather straightforward to understand.

The biggest part of the driver refers to the Gadget API.

20.13.4 Known Limitations

Like any other HW, DWC3 has its own set of limitations. To avoid constant questions about such problems, we decided to document them here and have a single location to where we could point users.

OUT Transfer Size Requirements

According to Synopsys Databook, all OUT transfer TRBs[1] must have their size field set to a value which is integer divisible by the endpoint's wMaxPacketSize. This means that e.g. in order to receive a Mass Storage CBW[5], req->length must either be set to a value that's divisible by wMaxPacketSize (1024 on SuperSpeed, 512 on HighSpeed, etc), or DWC3 driver must add a Chained TRB pointing to a throw-away buffer for the remaining length. Without this, OUT transfers will **NOT** start.

Note that as of this writing, this won't be a problem because DWC3 is fully capable of appending a chained TRB for the remaining length and completely hide this detail from the gadget driver. It's still worth mentioning because this seems to be the largest source of queries about DWC3 and non-working transfers.

TRB Ring Size Limitation

We, currently, have a hard limit of 256 TRBs[1] per endpoint, with the last TRB being a Link TRB[2] pointing back to the first. This limit is arbitrary but it has the benefit of adding up to exactly 4096 bytes, or 1 Page.

DWC3 driver will try its best to cope with more than 255 requests and, for the most part, it should work normally. However this is not something that has been exercised very frequently. If you experience any problems, see section **Reporting Bugs** below.

[5] Command Block Wrapper

[2] Transfer Request Block pointing to another Transfer Request Block.

20.13.5 Reporting Bugs

Whenever you encounter a problem with DWC3, first and foremost you should make sure that:

1. You're running latest tag from Linus' tree
2. You can reproduce the error without any out-of-tree changes to DWC3
3. You have checked that it's not a fault on the host machine

After all these are verified, then here's how to capture enough information so we can be of any help to you.

Required Information

DWC3 relies exclusively on Trace Events for debugging. Everything is exposed there, with some extra bits being exposed to DebugFS[3].

In order to capture DWC3's Trace Events you should run the following commands **before** plugging the USB cable to a host machine:

```
# mkdir -p /d
# mkdir -p /t
# mount -t debugfs none /d
# mount -t tracefs none /t
# echo 81920 > /t/buffer_size_kb
# echo 1 > /t/events/dwc3/enable
```

After this is done, you can connect your USB cable and reproduce the problem. As soon as the fault is reproduced, make a copy of files `trace` and `regdump`, like so:

```
# cp /t/trace /root/trace.txt
# cat /d/*dwc3*/regdump > /root/regdump.txt
```

Make sure to compress `trace.txt` and `regdump.txt` in a tarball and email it to me with linux-usb in Cc. If you want to be extra sure that I'll help you, write your subject line in the following format:

> **[BUG REPORT] usb: dwc3: Bug while doing XYZ**

On the email body, make sure to detail what you doing, which gadget driver you were using, how to reproduce the problem, what SoC you're using, which OS (and its version) was running on the Host machine.

With all this information, we should be able to understand what's going on and be helpful to you.

20.13.6 Debugging

First and foremost a disclaimer:

```
DISCLAIMER: The information available on DebugFS and/or TraceFS can
change at any time at any Major Linux Kernel Release. If writing
scripts, do **NOT** assume information to be available in the
current format.
```

With that out of the way, let's carry on.

If you're willing to debug your own problem, you deserve a round of applause :-)

Anyway, there isn't much to say here other than Trace Events will be really helpful in figuring out issues with DWC3. Also, access to Synopsys Databook will be **really** valuable in this case.

A USB Sniffer can be helpful at times but it's not entirely required, there's a lot that can be understood without looking at the wire.

Feel free to email me and Cc linux-usb if you need any help.

DebugFS

`DebugFS` is very good for gathering snapshots of what's going on with DWC3 and/or any endpoint.

On DWC3's `DebugFS` directory, you will find the following files and directories:

`ep[0..15]{in,out}/` `link_state` `regdump` `testmode`

link_state

When read, `link_state` will print out one of `U0`, `U1`, `U2`, `U3`, `SS.Disabled`, `RX.Detect`, `SS.Inactive`, `Polling`, `Recovery`, `Hot Reset`, `Compliance`, `Loopback`, `Reset`, `Resume` or `UNKNOWN link state`.

This file can also be written to in order to force link to one of the states above.

regdump

File name is self-explanatory. When read, `regdump` will print out a register dump of DWC3. Note that this file can be grepped to find the information you want.

testmode

When read, `testmode` will print out a name of one of the specified USB 2.0 Test-modes (`test_j`, `test_k`, `test_se0_nak`, `test_packet`, `test_force_enable`) or the string `no test` in case no tests are currently being executed.

In order to start any of these test modes, the same strings can be written to the file and DWC3 will enter the requested test mode.

ep[0..15]{in,out}

For each endpoint we expose one directory following the naming convention `ep$num$dir` (ep0in, ep0out, ep1in, ⋯). Inside each of these directories you will find the following files:

`descriptor_fetch_queue` `event_queue` `rx_fifo_queue` `rx_info_queue` `rx_request_queue` `transfer_type` `trb_ring` `tx_fifo_queue` `tx_request_queue`

With access to Synopsys Databook, you can decode the information on them.

transfer_type

When read, `transfer_type` will print out one of `control`, `bulk`, `interrupt` or `isochronous` depending on what the endpoint descriptor says. If the endpoint hasn't been enabled yet, it will print `--`.

trb_ring

When read, `trb_ring` will print out details about all TRBs on the ring. It will also tell you where our enqueue and dequeue pointers are located in the ring:

```
buffer_addr,size,type,ioc,isp_imi,csp,chn,lst,hwo
000000002c754000,481,normal,1,0,1,0,0,0
000000002c75c000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c75c000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
```

(continues on next page)

(continued from previous page)

```
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c75c000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
```

(continues on next page)

(continued from previous page)

```
000000002c754000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c75c000,481,normal,1,0,1,0,0,0
000000002c780000,481,normal,1,0,1,0,0,0
000000002c784000,481,normal,1,0,1,0,0,0
000000002c788000,481,normal,1,0,1,0,0,0
000000002c78c000,481,normal,1,0,1,0,0,0
000000002c790000,481,normal,1,0,1,0,0,0
000000002c754000,481,normal,1,0,1,0,0,0
000000002c758000,481,normal,1,0,1,0,0,0
000000002c75c000,512,normal,1,0,1,0,0,1        D
0000000000000000,0,UNKNOWN,0,0,0,0,0,0       E
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
```

(continues on next page)

(continued from previous page)

```
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
```

(continues on next page)

(continued from previous page)

```
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
```

(continues on next page)

(continued from previous page)

```
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
0000000000000000,0,UNKNOWN,0,0,0,0,0,0
00000000381ab000,0,link,0,0,0,0,0,1
```

Trace Events

DWC3 also provides several trace events which help us gathering information about the behavior of the driver during runtime.

In order to use these events, you must enable `CONFIG_FTRACE` in your kernel config.

For details about how enable DWC3 events, see section **Reporting Bugs**.

The following subsections will give details about each Event Class and each Event defined by DWC3.

MMIO

It is sometimes useful to look at every MMIO access when looking for bugs. Because of that, DWC3 offers two Trace Events (one for dwc3_readl() and one for dwc3_writel()). `TP_printk` follows:

```
TP_printk("addr %p value %08x", __entry->base + __entry->offset,
              __entry->value)
```

Interrupt Events

Every IRQ event can be logged and decoded into a human readable string. Because every event will be different, we don’t give an example other than the `TP_printk` format used:

```
TP_printk("event (%08x): %s", __entry->event,
              dwc3_decode_event(__entry->event, __entry->ep0state))
```

Control Request

Every USB Control Request can be logged to the trace buffer. The output format is:

```
TP_printk("%s", dwc3_decode_ctrl(__entry->bRequestType,
                                 __entry->bRequest, __entry->wValue,
                                 __entry->wIndex, __entry->wLength)
)
```

Note that Standard Control Requests will be decoded into human-readable strings with their respective arguments. Class and Vendor requests will be printed out a sequence of 8 bytes in hex format.

Lifetime of a `struct usb_request`

The entire lifetime of a `struct usb_request` can be tracked on the trace buffer. We have one event for each of allocation, free, queueing, dequeueing, and giveback. Output format is:

```
TP_printk("%s: req %p length %u/%u %s%s%s ==> %d",
      __get_str(name), __entry->req, __entry->actual, __entry->length,
      __entry->zero ? "Z" : "z",
      __entry->short_not_ok ? "S" : "s",
      __entry->no_interrupt ? "i" : "I",
      __entry->status
)
```

Generic Commands

We can log and decode every Generic Command with its completion code. Format is:

```
TP_printk("cmd '%s' [%x] param %08x --> status: %s",
      dwc3_gadget_generic_cmd_string(__entry->cmd),
      __entry->cmd, __entry->param,
      dwc3_gadget_generic_cmd_status_string(__entry->status)
)
```

Endpoint Commands

Endpoints commands can also be logged together with completion code. Format is:

```
TP_printk("%s: cmd '%s' [%d] params %08x %08x %08x --> status: %s",
      __get_str(name), dwc3_gadget_ep_cmd_string(__entry->cmd),
      __entry->cmd, __entry->param0,
      __entry->param1, __entry->param2,
      dwc3_ep_cmd_status_string(__entry->cmd_status)
)
```

Lifetime of a TRB

A TRB Lifetime is simple. We are either preparing a TRB or completing it. With these two events, we can see how a TRB changes over time. Format is:

```
TP_printk("%s: %d/%d trb %p buf %08x%08x size %s%d ctrl %08x (%c%c%c%c:%c
↪%c:%s)",
      __get_str(name), __entry->queued, __entry->allocated,
      __entry->trb, __entry->bph, __entry->bpl,
      ({char *s;
      int pcm = ((__entry->size >> 24) & 3) + 1;
      switch (__entry->type) {
      case USB_ENDPOINT_XFER_INT:
      case USB_ENDPOINT_XFER_ISOC:
              switch (pcm) {
              case 1:
                      s = "1x ";
                      break;
              case 2:
                      s = "2x ";
                      break;
              case 3:
                      s = "3x ";
                      break;
              }
      default:
              s = "";
      } s; }),
      DWC3_TRB_SIZE_LENGTH(__entry->size), __entry->ctrl,
      __entry->ctrl & DWC3_TRB_CTRL_HWO ? 'H' : 'h',
      __entry->ctrl & DWC3_TRB_CTRL_LST ? 'L' : 'l',
      __entry->ctrl & DWC3_TRB_CTRL_CHN ? 'C' : 'c',
      __entry->ctrl & DWC3_TRB_CTRL_CSP ? 'S' : 's',
      __entry->ctrl & DWC3_TRB_CTRL_ISP_IMI ? 'S' : 's',
      __entry->ctrl & DWC3_TRB_CTRL_IOC ? 'C' : 'c',
    dwc3_trb_type_string(DWC3_TRBCTL_TYPE(__entry->ctrl))
)
```

Lifetime of an Endpoint

And endpoint’s lifetime is summarized with enable and disable operations, both of which can be traced. Format is:

```
TP_printk("%s: mps %d/%d streams %d burst %d ring %d/%d flags %c:%c%c%c%c
↪%c:%c:%c",
      __get_str(name), __entry->maxpacket,
      __entry->maxpacket_limit, __entry->max_streams,
      __entry->maxburst, __entry->trb_enqueue,
      __entry->trb_dequeue,
      __entry->flags & DWC3_EP_ENABLED ? 'E' : 'e',
      __entry->flags & DWC3_EP_STALL ? 'S' : 's',
      __entry->flags & DWC3_EP_WEDGE ? 'W' : 'w',
      __entry->flags & DWC3_EP_TRANSFER_STARTED ? 'B' : 'b',
      __entry->flags & DWC3_EP_PENDING_REQUEST ? 'P' : 'p',
```

(continues on next page)

(continued from previous page)

```
      __entry->flags & DWC3_EP_END_TRANSFER_PENDING ? 'E' : 'e',
      __entry->direction ? '<' : '>'
)
```

20.13.7 Structures, Methods and Definitions

struct **dwc3_event_buffer**
Software event buffer representation

Definition

```
struct dwc3_event_buffer {
  void *buf;
  void *cache;
  unsigned length;
  unsigned int            lpos;
  unsigned int            count;
  unsigned int            flags;
#define DWC3_EVENT_PENDING      BIT(0);
  dma_addr_t dma;
  struct dwc3             *dwc;
};
```

Members

buf _THE_ buffer

cache The buffer cache used in the threaded interrupt

length size of this buffer

lpos event offset

count cache of last read event count register

flags flags related to this event buffer

dma dma_addr_t

dwc pointer to DWC controller

struct **dwc3_ep**
device side endpoint representation

Definition

```
struct dwc3_ep {
  struct usb_ep           endpoint;
  struct list_head        cancelled_list;
  struct list_head        pending_list;
  struct list_head        started_list;
  void __iomem            *regs;
  struct dwc3_trb         *trb_pool;
  dma_addr_t trb_pool_dma;
  struct dwc3             *dwc;
  u32 saved_state;
  unsigned flags;
```

(continues on next page)

(continued from previous page)

```
#define DWC3_EP_ENABLED         BIT(0);
#define DWC3_EP_STALL           BIT(1);
#define DWC3_EP_WEDGE           BIT(2);
#define DWC3_EP_TRANSFER_STARTED BIT(3);
#define DWC3_EP_END_TRANSFER_PENDING BIT(4);
#define DWC3_EP_PENDING_REQUEST BIT(5);
#define DWC3_EP_DELAY_START     BIT(6);
#define DWC3_EP_WAIT_TRANSFER_COMPLETE  BIT(7);
#define DWC3_EP_IGNORE_NEXT_NOSTREAM    BIT(8);
#define DWC3_EP_FORCE_RESTART_STREAM    BIT(9);
#define DWC3_EP_FIRST_STREAM_PRIMED     BIT(10);
#define DWC3_EP0_DIR_IN         BIT(31);
  u8 trb_enqueue;
  u8 trb_dequeue;
  u8 number;
  u8 type;
  u8 resource_index;
  u32 frame_number;
  u32 interval;
  char name[20];
  unsigned direction:1;
  unsigned stream_capable:1;
  u8 combo_num;
  int start_cmd_status;
};
```

Members

endpoint usb endpoint

cancelled_list list of cancelled requests for this endpoint

pending_list list of pending requests for this endpoint

started_list list of started requests on this endpoint

regs pointer to first endpoint register

trb_pool array of transaction buffers

trb_pool_dma dma address of **trb_pool**

dwc pointer to DWC controller

saved_state ep state saved during hibernation

flags endpoint flags (wedged, stalled, ⋯)

trb_enqueue enqueue ‘pointer’ into TRB array

trb_dequeue dequeue ‘pointer’ into TRB array

number endpoint number (1 - 15)

type set to bmAttributes & USB_ENDPOINT_XFERTYPE_MASK

resource_index Resource transfer index

frame_number set to the frame number we want this transfer to start (ISOC)

interval the interval on which the ISOC transfer is started

name a human readable name e.g. ep1out-bulk

direction true for TX, false for RX

stream_capable true when streams are enabled

combo_num the test combination BIT[15:14] of the frame number to test isochronous START TRANSFER command failure workaround

start_cmd_status the status of testing START TRANSFER command with combo_num = ‘b00

struct **dwc3_trb**
transfer request block (hw format)

Definition

```
struct dwc3_trb {
  u32 bpl;
  u32 bph;
  u32 size;
  u32 ctrl;
};
```

Members

bpl DW0-3

bph DW4-7

size DW8-B

ctrl DWC-F

struct **dwc3_hwparams**
copy of HWPARAMS registers

Definition

```
struct dwc3_hwparams {
  u32 hwparams0;
  u32 hwparams1;
  u32 hwparams2;
  u32 hwparams3;
  u32 hwparams4;
  u32 hwparams5;
  u32 hwparams6;
  u32 hwparams7;
  u32 hwparams8;
};
```

Members

hwparams0 GHWPARAMS0

hwparams1 GHWPARAMS1

hwparams2 GHWPARAMS2

hwparams3 GHWPARAMS3

hwparams4 GHWPARAMS4

hwparams5 GHWPARAMS5

hwparams6 GHWPARAMS6

hwparams7 GHWPARAMS7

hwparams8 GHWPARAMS8

struct **dwc3_request**
representation of a transfer request

Definition

```
struct dwc3_request {
  struct usb_request      request;
  struct list_head        list;
  struct dwc3_ep          *dep;
  struct scatterlist      *sg;
  struct scatterlist      *start_sg;
  unsigned num_pending_sgs;
  unsigned int            num_queued_sgs;
  unsigned remaining;
  unsigned int            status;
#define DWC3_REQUEST_STATUS_QUEUED      0;
#define DWC3_REQUEST_STATUS_STARTED     1;
#define DWC3_REQUEST_STATUS_CANCELLED   2;
#define DWC3_REQUEST_STATUS_COMPLETED   3;
#define DWC3_REQUEST_STATUS_UNKNOWN     -1;
  u8 epnum;
  struct dwc3_trb         *trb;
  dma_addr_t trb_dma;
  unsigned num_trbs;
  unsigned needs_extra_trb:1;
  unsigned direction:1;
  unsigned mapped:1;
};
```

Members

request struct usb_request to be transferred

list a list_head used for request queueing

dep struct dwc3_ep owning this request

sg pointer to first incomplete sg

start_sg pointer to the sg which should be queued next

num_pending_sgs counter to pending sgs

num_queued_sgs counter to the number of sgs which already got queued

remaining amount of data remaining

status internal dwc3 request status tracking

epnum endpoint number to which this request refers

trb pointer to struct dwc3_trb

trb_dma DMA address of **trb**

num_trbs number of TRBs used by this request

needs_extra_trb true when request needs one extra TRB (either due to ZLP or unaligned OUT)

direction IN or OUT direction flag

mapped true when request has been dma-mapped

struct **dwc3**
representation of our controller

Definition

```
struct dwc3 {
  struct work_struct      drd_work;
  struct dwc3_trb         *ep0_trb;
  void *bounce;
  void *scratchbuf;
  u8 *setup_buf;
  dma_addr_t ep0_trb_addr;
  dma_addr_t bounce_addr;
  dma_addr_t scratch_addr;
  struct dwc3_request     ep0_usb_req;
  struct completion       ep0_in_setup;
  spinlock_t lock;
  struct device           *dev;
  struct device           *sysdev;
  struct platform_device  *xhci;
  struct resource         xhci_resources[DWC3_XHCI_RESOURCES_NUM];
  struct dwc3_event_buffer *ev_buf;
  struct dwc3_ep          *eps[DWC3_ENDPOINTS_NUM];
  struct usb_gadget       gadget;
  struct usb_gadget_driver *gadget_driver;
  struct clk_bulk_data    *clks;
  int num_clks;
  struct reset_control    *reset;
  struct usb_phy          *usb2_phy;
  struct usb_phy          *usb3_phy;
  struct phy              *usb2_generic_phy;
  struct phy              *usb3_generic_phy;
  bool phys_ready;
  struct ulpi             *ulpi;
  bool ulpi_ready;
  void __iomem            *regs;
  size_t regs_size;
  enum usb_dr_mode        dr_mode;
  u32 current_dr_role;
  u32 desired_dr_role;
  struct extcon_dev       *edev;
  struct notifier_block   edev_nb;
  enum usb_phy_interface  hsphy_mode;
  struct usb_role_switch  *role_sw;
  enum usb_dr_mode        role_switch_default_mode;
  u32 fladj;
  u32 irq_gadget;
  u32 otg_irq;
  u32 current_otg_role;
```

(continues on next page)

(continued from previous page)

```
  u32 desired_otg_role;
  bool otg_restart_host;
  u32 nr_scratch;
  u32 u1u2;
  u32 maximum_speed;
  u32 ip;
#define DWC3_IP                 0x5533;
#define DWC31_IP                0x3331;
#define DWC32_IP                0x3332;
  u32 revision;
#define DWC3_REVISION_ANY       0x0;
#define DWC3_REVISION_173A      0x5533173a;
#define DWC3_REVISION_175A      0x5533175a;
#define DWC3_REVISION_180A      0x5533180a;
#define DWC3_REVISION_183A      0x5533183a;
#define DWC3_REVISION_185A      0x5533185a;
#define DWC3_REVISION_187A      0x5533187a;
#define DWC3_REVISION_188A      0x5533188a;
#define DWC3_REVISION_190A      0x5533190a;
#define DWC3_REVISION_194A      0x5533194a;
#define DWC3_REVISION_200A      0x5533200a;
#define DWC3_REVISION_202A      0x5533202a;
#define DWC3_REVISION_210A      0x5533210a;
#define DWC3_REVISION_220A      0x5533220a;
#define DWC3_REVISION_230A      0x5533230a;
#define DWC3_REVISION_240A      0x5533240a;
#define DWC3_REVISION_250A      0x5533250a;
#define DWC3_REVISION_260A      0x5533260a;
#define DWC3_REVISION_270A      0x5533270a;
#define DWC3_REVISION_280A      0x5533280a;
#define DWC3_REVISION_290A      0x5533290a;
#define DWC3_REVISION_300A      0x5533300a;
#define DWC3_REVISION_310A      0x5533310a;
#define DWC3_REVISION_330A      0x5533330a;
#define DWC31_REVISION_ANY      0x0;
#define DWC31_REVISION_110A     0x3131302a;
#define DWC31_REVISION_120A     0x3132302a;
#define DWC31_REVISION_160A     0x3136302a;
#define DWC31_REVISION_170A     0x3137302a;
#define DWC31_REVISION_180A     0x3138302a;
#define DWC31_REVISION_190A     0x3139302a;
#define DWC32_REVISION_ANY      0x0;
#define DWC32_REVISION_100A     0x3130302a;
  u32 version_type;
#define DWC31_VERSIONTYPE_ANY           0x0;
#define DWC31_VERSIONTYPE_EA01          0x65613031;
#define DWC31_VERSIONTYPE_EA02          0x65613032;
#define DWC31_VERSIONTYPE_EA03          0x65613033;
#define DWC31_VERSIONTYPE_EA04          0x65613034;
#define DWC31_VERSIONTYPE_EA05          0x65613035;
#define DWC31_VERSIONTYPE_EA06          0x65613036;
  enum dwc3_ep0_next      ep0_next_event;
  enum dwc3_ep0_state     ep0state;
  enum dwc3_link_state    link_state;
  u16 u2sel;
  u16 u2pel;
```

(continues on next page)

(continued from previous page)

```
  u8 u1sel;
  u8 u1pel;
  u8 speed;
  u8 num_eps;
  struct dwc3_hwparams    hwparams;
  struct dentry           *root;
  struct debugfs_regset32 *regset;
  u32 dbg_lsp_select;
  u8 test_mode;
  u8 test_mode_nr;
  u8 lpm_nyet_threshold;
  u8 hird_threshold;
  u8 rx_thr_num_pkt_prd;
  u8 rx_max_burst_prd;
  u8 tx_thr_num_pkt_prd;
  u8 tx_max_burst_prd;
  const char              *hsphy_interface;
  unsigned connected:1;
  unsigned delayed_status:1;
  unsigned ep0_bounced:1;
  unsigned ep0_expect_in:1;
  unsigned has_hibernation:1;
  unsigned sysdev_is_parent:1;
  unsigned has_lpm_erratum:1;
  unsigned is_utmi_l1_suspend:1;
  unsigned is_fpga:1;
  unsigned pending_events:1;
  unsigned pullups_connected:1;
  unsigned setup_packet_pending:1;
  unsigned three_stage_setup:1;
  unsigned dis_start_transfer_quirk:1;
  unsigned usb3_lpm_capable:1;
  unsigned usb2_lpm_disable:1;
  unsigned disable_scramble_quirk:1;
  unsigned u2exit_lfps_quirk:1;
  unsigned u2ss_inp3_quirk:1;
  unsigned req_p1p2p3_quirk:1;
  unsigned del_p1p2p3_quirk:1;
  unsigned del_phy_power_chg_quirk:1;
  unsigned lfps_filter_quirk:1;
  unsigned rx_detect_poll_quirk:1;
  unsigned dis_u3_susphy_quirk:1;
  unsigned dis_u2_susphy_quirk:1;
  unsigned dis_enblslpm_quirk:1;
  unsigned dis_u1_entry_quirk:1;
  unsigned dis_u2_entry_quirk:1;
  unsigned dis_rxdet_inp3_quirk:1;
  unsigned dis_u2_freeclk_exists_quirk:1;
  unsigned dis_del_phy_power_chg_quirk:1;
  unsigned dis_tx_ipgap_linecheck_quirk:1;
  unsigned parkmode_disable_ss_quirk:1;
  unsigned tx_de_emphasis_quirk:1;
  unsigned tx_de_emphasis:2;
  unsigned dis_metastability_quirk:1;
  u16 imod_interval;
};
```

Members

drd_work workqueue used for role swapping

ep0_trb trb which is used for the ctrl_req

bounce address of bounce buffer

scratchbuf address of scratch buffer

setup_buf used while precessing STD USB requests

ep0_trb_addr dma address of **ep0_trb**

bounce_addr dma address of **bounce**

scratch_addr dma address of scratchbuf

ep0_usb_req dummy req used while handling STD USB requests

ep0_in_setup one control transfer is completed and enter setup phase

lock for synchronizing

dev pointer to our struct device

sysdev pointer to the DMA-capable device

xhci pointer to our xHCI child

xhci_resources struct resources for our **xhci** child

ev_buf struct dwc3_event_buffer pointer

eps endpoint array

gadget device side representation of the peripheral controller

gadget_driver pointer to the gadget driver

clks array of clocks

num_clks number of clocks

reset reset control

usb2_phy pointer to USB2 PHY

usb3_phy pointer to USB3 PHY

usb2_generic_phy pointer to USB2 PHY

usb3_generic_phy pointer to USB3 PHY

phys_ready flag to indicate that PHYs are ready

ulpi pointer to ulpi interface

ulpi_ready flag to indicate that ULPI is initialized

regs base address for our registers

regs_size address space size

dr_mode requested mode of operation

current_dr_role current role of operation when in dual-role mode

desired_dr_role desired role of operation when in dual-role mode

edev extcon handle

edev_nb extcon notifier

hsphy_mode UTMI phy mode, one of following: - USBPHY_INTERFACE_MODE_UTMI - USBPHY_INTERFACE_MODE_UTMIW

role_sw usb_role_switch handle

role_switch_default_mode default operation mode of controller while usb role is USB_ROLE_NONE.

fladj frame length adjustment

irq_gadget peripheral controller's IRQ number

otg_irq IRQ number for OTG IRQs

current_otg_role current role of operation while using the OTG block

desired_otg_role desired role of operation while using the OTG block

otg_restart_host flag that OTG controller needs to restart host

nr_scratch number of scratch buffers

u1u2 only used on revisions <1.83a for workaround

maximum_speed maximum speed requested (mainly for testing purposes)

ip controller's ID

revision controller's version of an IP

version_type VERSIONTYPE register contents, a sub release of a revision

ep0_next_event hold the next expected event

ep0state state of endpoint zero

link_state link state

u2sel parameter from Set SEL request.

u2pel parameter from Set SEL request.

u1sel parameter from Set SEL request.

u1pel parameter from Set SEL request.

speed device speed (super, high, full, low)

num_eps number of endpoints

hwparams copy of hwparams registers

root debugfs root folder pointer

regset debugfs pointer to regdump file

dbg_lsp_select current debug lsp mux register selection

test_mode true when we're entering a USB test mode

test_mode_nr test feature selector

lpm_nyet_threshold LPM NYET response threshold

hird_threshold HIRD threshold

rx_thr_num_pkt_prd periodic ESS receive packet count

rx_max_burst_prd max periodic ESS receive burst size

tx_thr_num_pkt_prd periodic ESS transmit packet count

tx_max_burst_prd max periodic ESS transmit burst size

hsphy_interface "utmi" or "ulpi"

connected true when we're connected to a host, false otherwise

delayed_status true when gadget driver asks for delayed status

ep0_bounced true when we used bounce buffer

ep0_expect_in true when we expect a DATA IN transfer

has_hibernation true when dwc3 was configured with Hibernation

sysdev_is_parent true when dwc3 device has a parent driver

has_lpm_erratum true when core was configured with LPM Erratum. Note that there's now way for software to detect this in runtime.

is_utmi_l1_suspend the core asserts output signal 0 - utmi_sleep_n 1 - utmi_l1_suspend_n

is_fpga true when we are using the FPGA board

pending_events true when we have pending IRQs to be handled

pullups_connected true when Run/Stop bit is set

setup_packet_pending true when there's a Setup Packet in FIFO. Workaround

three_stage_setup set if we perform a three phase setup

dis_start_transfer_quirk set if start_transfer failure SW workaround is not needed for DWC_usb31 version 1.70a-ea06 and below

usb3_lpm_capable set if hadrware supports Link Power Management

usb2_lpm_disable set to disable usb2 lpm

disable_scramble_quirk set if we enable the disable scramble quirk

u2exit_lfps_quirk set if we enable u2exit lfps quirk

u2ss_inp3_quirk set if we enable P3 OK for U2/SS Inactive quirk

req_p1p2p3_quirk set if we enable request p1p2p3 quirk

del_p1p2p3_quirk set if we enable delay p1p2p3 quirk

del_phy_power_chg_quirk set if we enable delay phy power change quirk

lfps_filter_quirk set if we enable LFPS filter quirk

rx_detect_poll_quirk set if we enable rx_detect to polling lfps quirk

dis_u3_susphy_quirk set if we disable usb3 suspend phy

dis_u2_susphy_quirk set if we disable usb2 suspend phy

dis_enblslpm_quirk set if we clear enblslpm in GUSB2PHYCFG, disabling the suspend signal to the PHY.

dis_u1_entry_quirk set if link entering into U1 state needs to be disabled.

dis_u2_entry_quirk set if link entering into U2 state needs to be disabled.

dis_rxdet_inp3_quirk set if we disable Rx.Detect in P3

dis_u2_freeclk_exists_quirk set if we clear u2_freeclk_exists in GUSB2PHYCFG, specify that USB2 PHY doesn't provide a free-running PHY clock.

dis_del_phy_power_chg_quirk set if we disable delay phy power change quirk.

dis_tx_ipgap_linecheck_quirk set if we disable u2mac linestate check during HS transmit.

parkmode_disable_ss_quirk set if we need to disable all SuperSpeed instances in park mode.

tx_de_emphasis_quirk set if we enable Tx de-emphasis quirk

tx_de_emphasis Tx de-emphasis value 0 - -6dB de-emphasis 1 - -3.5dB de-emphasis 2 - No de-emphasis 3 - Reserved

dis_metastability_quirk set to disable metastability quirk.

imod_interval set the interrupt moderation interval in 250ns increments or 0 to disable.

struct **dwc3_event_depevt**
Device Endpoint Events

Definition

```
struct dwc3_event_depevt {
  u32 one_bit:1;
  u32 endpoint_number:5;
  u32 endpoint_event:4;
  u32 reserved11_10:2;
  u32 status:4;
#define DEPEVT_STATUS_TRANSFER_ACTIVE   BIT(3);
#define DEPEVT_STATUS_BUSERR    BIT(0);
#define DEPEVT_STATUS_SHORT     BIT(1);
#define DEPEVT_STATUS_IOC       BIT(2);
#define DEPEVT_STATUS_LST       BIT(3) ;
#define DEPEVT_STATUS_MISSED_ISOC BIT(3) ;
#define DEPEVT_STREAMEVT_FOUND          1;
#define DEPEVT_STREAMEVT_NOTFOUND       2;
#define DEPEVT_STREAM_PRIME             0xfffe;
#define DEPEVT_STREAM_NOSTREAM          0x0;
#define DEPEVT_STATUS_CONTROL_DATA      1;
#define DEPEVT_STATUS_CONTROL_STATUS    2;
#define DEPEVT_STATUS_CONTROL_PHASE(n)  ((n) & 3);
#define DEPEVT_TRANSFER_NO_RESOURCE     1;
#define DEPEVT_TRANSFER_BUS_EXPIRY      2;
  u32 parameters:16;
```

(continues on next page)

(continued from previous page)

```
#define DEPEVT_PARAMETER_CMD(n) (((n) & (0xf << 8)) >> 8);
};
```

Members

one_bit indicates this is an endpoint event (not used)

endpoint_number number of the endpoint

endpoint_event The event we have: 0x00 - Reserved 0x01 - XferComplete 0x02 - XferInProgress 0x03 - XferNotReady 0x04 - RxTxFifoEvt (IN->Underrun, OUT->Overrun) 0x05 - Reserved 0x06 - StreamEvt 0x07 - EPCmdCmplt

reserved11_10 Reserved, don't use.

status Indicates the status of the event. Refer to databook for more information.

parameters Parameters of the current event. Refer to databook for more information.

struct **dwc3_event_devt**
Device Events

Definition

```
struct dwc3_event_devt {
  u32 one_bit:1;
  u32 device_event:7;
  u32 type:4;
  u32 reserved15_12:4;
  u32 event_info:9;
  u32 reserved31_25:7;
};
```

Members

one_bit indicates this is a non-endpoint event (not used)

device_event indicates it's a device event. Should read as 0x00

type indicates the type of device event. 0 - DisconnEvt 1 - USBRst 2 - ConnectDone 3 - ULStChng 4 - WkUpEvt 5 - Reserved 6 - EOPF 7 - SOF 8 - Reserved 9 - ErrticErr 10 - CmdCmplt 11 - EvntOverflow 12 - VndrDevTstRcved

reserved15_12 Reserved, not used

event_info Information about this event

reserved31_25 Reserved, not used

struct **dwc3_event_gevt**
Other Core Events

Definition

```
struct dwc3_event_gevt {
  u32 one_bit:1;
  u32 device_event:7;
  u32 phy_port_number:4;
```

(continues on next page)

(continued from previous page)

```
  u32 reserved31_12:20;
};
```

Members

one_bit indicates this is a non-endpoint event (not used)

device_event indicates it’s (0x03) Carkit or (0x04) I2C event.

phy_port_number self-explanatory

reserved31_12 Reserved, not used.

union **dwc3_event**
representation of Event Buffer contents

Definition

```
union dwc3_event {
  u32 raw;
  struct dwc3_event_type          type;
  struct dwc3_event_depevt        depevt;
  struct dwc3_event_devt          devt;
  struct dwc3_event_gevt          gevt;
};
```

Members

raw raw 32-bit event

type the type of the event

depevt Device Endpoint Event

devt Device Event

gevt Global Event

struct **dwc3_gadget_ep_cmd_params**
representation of endpoint command parameters

Definition

```
struct dwc3_gadget_ep_cmd_params {
  u32 param2;
  u32 param1;
  u32 param0;
};
```

Members

param2 third parameter

param1 second parameter

param0 first parameter

struct dwc3_request * **next_request**(struct list_head * list)
gets the next request on the given list

Parameters

struct list_head * list the request list to operate on

Description

Caller should take care of locking. This function return NULL or the first request available on **list**.

void **dwc3_gadget_move_started_request**(struct dwc3_request * req)
move **req** to the started_list

Parameters

struct dwc3_request * req the request to be moved

Description

Caller should take care of locking. This function will move **req** from its current list to the endpoint's started_list.

void **dwc3_gadget_move_cancelled_request**(struct dwc3_request * req)
move **req** to the cancelled_list

Parameters

struct dwc3_request * req the request to be moved

Description

Caller should take care of locking. This function will move **req** from its current list to the endpoint's cancelled_list.

void **dwc3_gadget_ep_get_transfer_index**(struct dwc3_ep * dep)
Gets transfer index from HW

Parameters

struct dwc3_ep * dep dwc3 endpoint

Description

Caller should take care of locking. Returns the transfer resource index for a given endpoint.

void **dwc3_gadget_dctl_write_safe**(struct dwc3 * dwc, u32 value)
write to DCTL safe from link state change

Parameters

struct dwc3 * dwc pointer to our context structure

u32 value value to write to DCTL

Description

Use this function when doing read-modify-write to DCTL. It will not send link state change request.

int **dwc3_gadget_set_test_mode**(struct dwc3 * dwc, int mode)
enables usb2 test modes

Parameters

struct dwc3 * dwc pointer to our context structure

int mode the mode to set (J, K SE0 NAK, Force Enable)

Description

Caller should take care of locking. This function will return 0 on success or -EINVAL if wrong Test Selector is passed.

int **dwc3_gadget_get_link_state**(struct dwc3 * dwc)
gets current state of usb link

Parameters

struct dwc3 * dwc pointer to our context structure

Description

Caller should take care of locking. This function will return the link state on success (>= 0) or -ETIMEDOUT.

int **dwc3_gadget_set_link_state**(struct dwc3 * dwc, enum dwc3_link_state state)
sets usb link to a particular state

Parameters

struct dwc3 * dwc pointer to our context structure

enum dwc3_link_state state the state to put link into

Description

Caller should take care of locking. This function will return 0 on success or -ETIMEDOUT.

void **dwc3_ep_inc_trb**(u8 * index)
increment a trb index.

Parameters

u8 * index Pointer to the TRB index to increment.

Description

The index should never point to the link TRB. After incrementing, if it is point to the link TRB, wrap around to the beginning. The link TRB is always at the last TRB entry.

void **dwc3_ep_inc_enq**(struct dwc3_ep * dep)
increment endpoint's enqueue pointer

Parameters

struct dwc3_ep * dep The endpoint whose enqueue pointer we're incrementing

void **dwc3_ep_inc_deq**(struct dwc3_ep * dep)
increment endpoint's dequeue pointer

Parameters

struct dwc3_ep * dep The endpoint whose enqueue pointer we're incrementing

void **dwc3_gadget_giveback**(struct dwc3_ep * dep, struct dwc3_request * req, int status)
call struct usb_request's ->complete callback

Parameters

struct dwc3_ep * dep The endpoint to whom the request belongs to

struct dwc3_request * req The request we' re giving back

int status completion code for the request

Description

Must be called with controller' s lock held and interrupts disabled. This function will unmap **req** and call its ->complete() callback to notify upper layers that it has completed.

int **dwc3_send_gadget_generic_command**(struct dwc3 * dwc, unsigned cmd, u32 param)
issue a generic command for the controller

Parameters

struct dwc3 * dwc pointer to the controller context

unsigned cmd the command to be issued

u32 param command parameter

Description

Caller should take care of locking. Issue **cmd** with a given **param** to **dwc** and wait for its completion.

int **dwc3_send_gadget_ep_cmd**(struct dwc3_ep * dep, unsigned cmd, struct dwc3_gadget_ep_cmd_params * params)
issue an endpoint command

Parameters

struct dwc3_ep * dep the endpoint to which the command is going to be issued

unsigned cmd the command to be issued

struct dwc3_gadget_ep_cmd_params * params parameters to the command

Description

Caller should handle locking. This function will issue **cmd** with given **params** to **dep** and wait for its completion.

int **dwc3_gadget_start_config**(struct dwc3_ep * dep)
configure ep resources

Parameters

struct dwc3_ep * dep endpoint that is being enabled

Description

Issue a DWC3_DEPCMD_DEPSTARTCFG command to **dep**. After the command' s completion, it will set Transfer Resource for all available endpoints.

The assignment of transfer resources cannot perfectly follow the data book due to the fact that the controller driver does not have all knowledge of the configuration in advance. It is given this information piecemeal by the composite gadget framework after every SET_CONFIGURATION and SET_INTERFACE. Trying to follow

the databook programming model in this scenario can cause errors. For two reasons:

1) The databook says to do DWC3_DEPCMD_DEPSTARTCFG for every USB_REQ_SET_CONFIGURATION and USB_REQ_SET_INTERFACE (8.1.5). This is incorrect in the scenario of multiple interfaces.

2) The databook does not mention doing more DWC3_DEPCMD_DEPXFERCFG for new endpoint on alt setting (8.1.6).

The following simplified method is used instead:

All hardware endpoints can be assigned a transfer resource and this setting will stay persistent until either a core reset or hibernation. So whenever we do a DWC3_DEPCMD_DEPSTARTCFG``(0) we can go ahead and do ``DWC3_DEPCMD_DEPXFERCFG for every hardware endpoint as well. We are guaranteed that there are as many transfer resources as endpoints.

This function is called for each endpoint when it is being enabled but is triggered only when called for EP0-out, which always happens first, and which should only happen in one of the above conditions.

int **__dwc3_gadget_ep_enable**(struct dwc3_ep * dep, unsigned int action)
initializes a hw endpoint

Parameters

struct dwc3_ep * dep endpoint to be initialized

unsigned int action one of INIT, MODIFY or RESTORE

Description

Caller should take care of locking. Execute all necessary commands to initialize a HW endpoint so it can be used by a gadget driver.

int **__dwc3_gadget_ep_disable**(struct dwc3_ep * dep)
disables a hw endpoint

Parameters

struct dwc3_ep * dep the endpoint to disable

Description

This function undoes what __dwc3_gadget_ep_enable did and also removes requests which are currently being processed by the hardware and those which are not yet scheduled.

Caller should take care of locking.

struct dwc3_trb * **dwc3_ep_prev_trb**(struct dwc3_ep * dep, u8 index)
returns the previous TRB in the ring

Parameters

struct dwc3_ep * dep The endpoint with the TRB ring

u8 index The index of the current TRB in the ring

Description

Returns the TRB prior to the one pointed to by the index. If the index is 0, we will wrap backwards, skip the link TRB, and return the one just before that.

void **dwc3_prepare_one_trb**(struct dwc3_ep * dep, struct dwc3_request * req, unsigned chain, unsigned node)
setup one TRB from one request

Parameters

struct dwc3_ep * dep endpoint for which this request is prepared

struct dwc3_request * req dwc3_request pointer

unsigned chain should this TRB be chained to the next?

unsigned node only for isochronous endpoints. First TRB needs different type.

int **dwc3_gadget_start_isoc_quirk**(struct dwc3_ep * dep)
workaround invalid frame number

Parameters

struct dwc3_ep * dep isoc endpoint

Description

This function tests for the correct combination of BIT[15:14] from the 16-bit microframe number reported by the XferNotReady event for the future frame number to start the isoc transfer.

In DWC_usb31 version 1.70a-ea06 and prior, for highspeed and fullspeed isochronous IN, BIT[15:14] of the 16-bit microframe number reported by the XferNotReady event are invalid. The driver uses this number to schedule the isochronous transfer and passes it to the START TRANSFER command. Because this number is invalid, the command may fail. If BIT[15:14] matches the internal 16-bit microframe, the START TRANSFER command will pass and the transfer will start at the scheduled time, if it is off by 1, the command will still pass, but the transfer will start 2 seconds in the future. For all other conditions, the START TRANSFER command will fail with bus-expiry.

In order to workaround this issue, we can test for the correct combination of BIT[15:14] by sending START TRANSFER commands with different values of BIT[15:14]: ‘b00, ‘b01, ‘b10, and ‘b11. Each combination is 2^14 uframe apart (or 2 seconds). 4 seconds into the future will result in a bus-expiry status. As the result, within the 4 possible combinations for BIT[15:14], there will be 2 successful and 2 failure START COMMAND status. One of the 2 successful command status will result in a 2-second delay start. The smaller BIT[15:14] value is the correct combination.

Since there are only 4 outcomes and the results are ordered, we can simply test 2 START TRANSFER commands with BIT[15:14] combinations ‘b00 and ‘b01 to deduce the smaller successful combination.

Let test0 = test status for combination ‘b00 and test1 = test status for ‘b01 of BIT[15:14]. The correct combination is as follow:

if test0 fails and test1 passes, BIT[15:14] is ‘b01 if test0 fails and test1 fails, BIT[15:14] is ‘b10 if test0 passes and test1 fails, BIT[15:14] is ‘b11 if test0 passes and test1 passes, BIT[15:14] is ‘b00

Synopsys STAR 9001202023: Wrong microframe number for isochronous IN endpoints.

void **dwc3_gadget_setup_nump**(struct dwc3 * dwc)
calculate and initialize NUMP field of DWC3_DCFG

Parameters

struct dwc3 * dwc pointer to our context structure

Description

The following looks like complex but it's actually very simple. In order to calculate the number of packets we can burst at once on OUT transfers, we're gonna use RxFIFO size.

To calculate RxFIFO size we need two numbers: MDWIDTH = size, in bits, of the internal memory bus RAM2_DEPTH = depth, in MDWIDTH, of internal RAM2 (where RxFIFO sits)

Given these two numbers, the formula is simple:

RxFIFO Size = (RAM2_DEPTH * MDWIDTH / 8) - 24 - 16;

24 bytes is for 3x SETUP packets 16 bytes is a clock domain crossing tolerance

Given RxFIFO Size, NUMP = RxFIFOSize / 1024;

int **dwc3_gadget_init**(struct dwc3 * dwc)
initializes gadget related registers

Parameters

struct dwc3 * dwc pointer to our controller context structure

Description

Returns 0 on success otherwise negative errno.

DWC3_DEFAULT_AUTOSUSPEND_DELAY()
DesignWare USB3 DRD Controller Core file

Parameters

Description

Copyright (C) 2010-2011 Texas Instruments Incorporated - http://www.ti.com

Authors: Felipe Balbi <balbiti.com**>,** Sebastian Andrzej Siewior <bigeasy**linutronix.de**>

int **dwc3_get_dr_mode**(struct dwc3 * dwc)
Validates and sets dr_mode

Parameters

struct dwc3 * dwc pointer to our context structure

int **dwc3_core_soft_reset**(struct dwc3 * dwc)
Issues core soft reset and PHY reset

Parameters

struct dwc3 * dwc pointer to our context structure

void **dwc3_free_one_event_buffer**(struct dwc3 * dwc, struct dwc3_event_buffer * evt)
Frees one event buffer

Parameters

struct dwc3 * dwc Pointer to our controller context structure

struct dwc3_event_buffer * evt Pointer to event buffer to be freed

struct dwc3_event_buffer * **dwc3_alloc_one_event_buffer**(struct dwc3 * dwc, unsigned length)
Allocates one event buffer structure

Parameters

struct dwc3 * dwc Pointer to our controller context structure

unsigned length size of the event buffer

Description

Returns a pointer to the allocated event buffer structure on success otherwise ERR_PTR(errno).

void **dwc3_free_event_buffers**(struct dwc3 * dwc)
frees all allocated event buffers

Parameters

struct dwc3 * dwc Pointer to our controller context structure

int **dwc3_alloc_event_buffers**(struct dwc3 * dwc, unsigned length)
Allocates **num** event buffers of size **length**

Parameters

struct dwc3 * dwc pointer to our controller context structure

unsigned length size of event buffer

Description

Returns 0 on success otherwise negative errno. In the error case, dwc may contain some buffers allocated but not all which were requested.

int **dwc3_event_buffers_setup**(struct dwc3 * dwc)
setup our allocated event buffers

Parameters

struct dwc3 * dwc pointer to our controller context structure

Description

Returns 0 on success otherwise negative errno.

int **dwc3_phy_setup**(struct dwc3 * dwc)
Configure USB PHY Interface of DWC3 Core

Parameters

struct dwc3 * dwc Pointer to our controller context structure

Description

Returns 0 on success. The USB PHY interfaces are configured but not initialized. The PHY interfaces and the PHYs get initialized together with the core in dwc3_core_init.

int **dwc3_core_init**(struct dwc3 * dwc)
Low-level initialization of DWC3 Core

Parameters

struct dwc3 * dwc Pointer to our controller context structure

Description

Returns 0 on success otherwise negative errno.

20.14 Writing a MUSB Glue Layer

Author Apelete Seketeli

20.14.1 Introduction

The Linux MUSB subsystem is part of the larger Linux USB subsystem. It provides support for embedded USB Device Controllers (UDC) that do not use Universal Host Controller Interface (UHCI) or Open Host Controller Interface (OHCI).

Instead, these embedded UDC rely on the USB On-the-Go (OTG) specification which they implement at least partially. The silicon reference design used in most cases is the Multipoint USB Highspeed Dual-Role Controller (MUSB HDRC) found in the Mentor Graphics Inventra™ design.

As a self-taught exercise I have written an MUSB glue layer for the Ingenic JZ4740 SoC, modelled after the many MUSB glue layers in the kernel source tree. This layer can be found at `drivers/usb/musb/jz4740.c`. In this documentation I will walk through the basics of the `jz4740.c` glue layer, explaining the different pieces and what needs to be done in order to write your own device glue layer.

20.14.2 Linux MUSB Basics

To get started on the topic, please read USB On-the-Go Basics (see Resources) which provides an introduction of USB OTG operation at the hardware level. A couple of wiki pages by Texas Instruments and Analog Devices also provide an overview of the Linux kernel MUSB configuration, albeit focused on some specific devices provided by these companies. Finally, getting acquainted with the USB specification at USB home page may come in handy, with practical instance provided through the Writing USB Device Drivers documentation (again, see Resources).

Linux USB stack is a layered architecture in which the MUSB controller hardware sits at the lowest. The MUSB controller driver abstract the MUSB controller hardware to the Linux USB stack:

```
    ------------------------
    |                      | <------- drivers/usb/gadget
    | Linux USB Core Stack | <------- drivers/usb/host
    |                      | <------- drivers/usb/core
    ------------------------
               ↕
   --------------------------
   |                        | <------ drivers/usb/musb/musb_gadget.c
   | MUSB Controller driver | <------ drivers/usb/musb/musb_host.c
   |                        | <------ drivers/usb/musb/musb_core.c
   --------------------------
               ↕
--------------------------------
| MUSB Platform Specific Driver |
|                               | <-- drivers/usb/musb/jz4740.c
|       aka "Glue Layer"        |
--------------------------------
               ↕
--------------------------------
|   MUSB Controller Hardware    |
--------------------------------
```

As outlined above, the glue layer is actually the platform specific code sitting in between the controller driver and the controller hardware.

Just like a Linux USB driver needs to register itself with the Linux USB subsystem, the MUSB glue layer needs first to register itself with the MUSB controller driver. This will allow the controller driver to know about which device the glue layer supports and which functions to call when a supported device is detected or released; remember we are talking about an embedded controller chip here, so no insertion or removal at run-time.

All of this information is passed to the MUSB controller driver through a `platform_driver` structure defined in the glue layer as:

```
static struct platform_driver jz4740_driver = {
    .probe      = jz4740_probe,
    .remove     = jz4740_remove,
    .driver     = {
        .name   = "musb-jz4740",
    },
};
```

The probe and remove function pointers are called when a matching device is detected and, respectively, released. The name string describes the device supported by this glue layer. In the current case it matches a platform_device structure declared in `arch/mips/jz4740/platform.c`. Note that we are not using device tree bindings here.

In order to register itself to the controller driver, the glue layer goes through a few steps, basically allocating the controller hardware resources and initialising a couple of circuits. To do so, it needs to keep track of the information used throughout these steps. This is done by defining a private `jz4740_glue` structure:

```
struct jz4740_glue {
```

(continues on next page)

(continued from previous page)

```
    struct device           *dev;
    struct platform_device  *musb;
    struct clk      *clk;
};
```

The dev and musb members are both device structure variables. The first one holds generic information about the device, since it's the basic device structure, and the latter holds information more closely related to the subsystem the device is registered to. The clk variable keeps information related to the device clock operation.

Let's go through the steps of the probe function that leads the glue layer to register itself to the controller driver.

Note: For the sake of readability each function will be split in logical parts, each part being shown as if it was independent from the others.

```
static int jz4740_probe(struct platform_device *pdev)
{
    struct platform_device      *musb;
    struct jz4740_glue      *glue;
    struct clk                      *clk;
    int             ret;

    glue = devm_kzalloc(&pdev->dev, sizeof(*glue), GFP_KERNEL);
    if (!glue)
        return -ENOMEM;

    musb = platform_device_alloc("musb-hdrc", PLATFORM_DEVID_AUTO);
    if (!musb) {
        dev_err(&pdev->dev, "failed to allocate musb device\n");
        return -ENOMEM;
    }

    clk = devm_clk_get(&pdev->dev, "udc");
    if (IS_ERR(clk)) {
        dev_err(&pdev->dev, "failed to get clock\n");
        ret = PTR_ERR(clk);
        goto err_platform_device_put;
    }

    ret = clk_prepare_enable(clk);
    if (ret) {
        dev_err(&pdev->dev, "failed to enable clock\n");
        goto err_platform_device_put;
    }

    musb->dev.parent        = &pdev->dev;

    glue->dev           = &pdev->dev;
    glue->musb          = musb;
    glue->clk           = clk;
```

(continues on next page)

(continued from previous page)

```
    return 0;

err_platform_device_put:
    platform_device_put(musb);
    return ret;
}
```

The first few lines of the probe function allocate and assign the glue, musb and clk variables. The GFP_KERNEL flag (line 8) allows the allocation process to sleep and wait for memory, thus being usable in a locking situation. The PLATFORM_DEVID_AUTO flag (line 12) allows automatic allocation and management of device IDs in order to avoid device namespace collisions with explicit IDs. With devm_clk_get() (line 18) the glue layer allocates the clock – the devm_ prefix indicates that clk_get() is managed: it automatically frees the allocated clock resource data when the device is released – and enable it.

Then comes the registration steps:

```
static int jz4740_probe(struct platform_device *pdev)
{
    struct musb_hdrc_platform_data  *pdata = &jz4740_musb_platform_data;

    pdata->platform_ops     = &jz4740_musb_ops;

    platform_set_drvdata(pdev, glue);

    ret = platform_device_add_resources(musb, pdev->resource,
                        pdev->num_resources);
    if (ret) {
        dev_err(&pdev->dev, "failed to add resources\n");
        goto err_clk_disable;
    }

    ret = platform_device_add_data(musb, pdata, sizeof(*pdata));
    if (ret) {
        dev_err(&pdev->dev, "failed to add platform_data\n");
        goto err_clk_disable;
    }

    return 0;

err_clk_disable:
    clk_disable_unprepare(clk);
err_platform_device_put:
    platform_device_put(musb);
    return ret;
}
```

The first step is to pass the device data privately held by the glue layer on to the controller driver through platform_set_drvdata() (line 7). Next is passing on the device resources information, also privately held at that point, through platform_device_add_resources() (line 9).

Finally comes passing on the platform specific data to the controller driver (line 16). Platform data will be discussed in Device Platform Data, but here we are look-

ing at the `platform_ops` function pointer (line 5) in `musb_hdrc_platform_data` structure (line 3). This function pointer allows the MUSB controller driver to know which function to call for device operation:

```
static const struct musb_platform_ops jz4740_musb_ops = {
    .init       = jz4740_musb_init,
    .exit       = jz4740_musb_exit,
};
```

Here we have the minimal case where only init and exit functions are called by the controller driver when needed. Fact is the JZ4740 MUSB controller is a basic controller, lacking some features found in other controllers, otherwise we may also have pointers to a few other functions like a power management function or a function to switch between OTG and non-OTG modes, for instance.

At that point of the registration process, the controller driver actually calls the init function:

```
static int jz4740_musb_init(struct musb *musb)
{
    musb->xceiv = usb_get_phy(USB_PHY_TYPE_USB2);
    if (!musb->xceiv) {
        pr_err("HS UDC: no transceiver configured\n");
        return -ENODEV;
    }

    /* Silicon does not implement ConfigData register.
     * Set dyn_fifo to avoid reading EP config from hardware.
     */
    musb->dyn_fifo = true;

    musb->isr = jz4740_musb_interrupt;

    return 0;
}
```

The goal of `jz4740_musb_init()` is to get hold of the transceiver driver data of the MUSB controller hardware and pass it on to the MUSB controller driver, as usual. The transceiver is the circuitry inside the controller hardware responsible for sending/receiving the USB data. Since it is an implementation of the physical layer of the OSI model, the transceiver is also referred to as PHY.

Getting hold of the `MUSB PHY` driver data is done with `usb_get_phy()` which returns a pointer to the structure containing the driver instance data. The next couple of instructions (line 12 and 14) are used as a quirk and to setup IRQ handling respectively. Quirks and IRQ handling will be discussed later in Device Quirks and Handling IRQs

```
static int jz4740_musb_exit(struct musb *musb)
{
    usb_put_phy(musb->xceiv);

    return 0;
}
```

Acting as the counterpart of init, the exit function releases the MUSB PHY driver

when the controller hardware itself is about to be released.

Again, note that init and exit are fairly simple in this case due to the basic set of features of the JZ4740 controller hardware. When writing an musb glue layer for a more complex controller hardware, you might need to take care of more processing in those two functions.

Returning from the init function, the MUSB controller driver jumps back into the probe function:

```
static int jz4740_probe(struct platform_device *pdev)
{
    ret = platform_device_add(musb);
    if (ret) {
        dev_err(&pdev->dev, "failed to register musb device\n");
        goto err_clk_disable;
    }

    return 0;

err_clk_disable:
    clk_disable_unprepare(clk);
err_platform_device_put:
    platform_device_put(musb);
    return ret;
}
```

This is the last part of the device registration process where the glue layer adds the controller hardware device to Linux kernel device hierarchy: at this stage, all known information about the device is passed on to the Linux USB core stack:

```
static int jz4740_remove(struct platform_device *pdev)
{
    struct jz4740_glue  *glue = platform_get_drvdata(pdev);

    platform_device_unregister(glue->musb);
    clk_disable_unprepare(glue->clk);

    return 0;
}
```

Acting as the counterpart of probe, the remove function unregister the MUSB controller hardware (line 5) and disable the clock (line 6), allowing it to be gated.

20.14.3 Handling IRQs

Additionally to the MUSB controller hardware basic setup and registration, the glue layer is also responsible for handling the IRQs:

```
static irqreturn_t jz4740_musb_interrupt(int irq, void *__hci)
{
    unsigned long   flags;
    irqreturn_t     retval = IRQ_NONE;
    struct musb     *musb = __hci;
```

(continues on next page)

(continued from previous page)

```
    spin_lock_irqsave(&musb->lock, flags);

    musb->int_usb = musb_readb(musb->mregs, MUSB_INTRUSB);
    musb->int_tx = musb_readw(musb->mregs, MUSB_INTRTX);
    musb->int_rx = musb_readw(musb->mregs, MUSB_INTRRX);

    /*
     * The controller is gadget only, the state of the host mode␣
↪IRQ bits is
     * undefined. Mask them to make sure that the musb driver␣
↪core will
     * never see them set
     */
    musb->int_usb &= MUSB_INTR_SUSPEND | MUSB_INTR_RESUME |
        MUSB_INTR_RESET | MUSB_INTR_SOF;

    if (musb->int_usb || musb->int_tx || musb->int_rx)
        retval = musb_interrupt(musb);

    spin_unlock_irqrestore(&musb->lock, flags);

    return retval;
}
```

Here the glue layer mostly has to read the relevant hardware registers and pass their values on to the controller driver which will handle the actual event that triggered the IRQ.

The interrupt handler critical section is protected by the `spin_lock_irqsave()` and counterpart `spin_unlock_irqrestore()` functions (line 7 and 24 respectively), which prevent the interrupt handler code to be run by two different threads at the same time.

Then the relevant interrupt registers are read (line 9 to 11):

- `MUSB_INTRUSB`: indicates which USB interrupts are currently active,
- `MUSB_INTRTX`: indicates which of the interrupts for TX endpoints are currently active,
- `MUSB_INTRRX`: indicates which of the interrupts for TX endpoints are currently active.

Note that `musb_readb()` is used to read 8-bit registers at most, while `musb_readw()` allows us to read at most 16-bit registers. There are other functions that can be used depending on the size of your device registers. See `musb_io.h` for more information.

Instruction on line 18 is another quirk specific to the JZ4740 USB device controller, which will be discussed later in Device Quirks.

The glue layer still needs to register the IRQ handler though. Remember the instruction on line 14 of the init function:

```
static int jz4740_musb_init(struct musb *musb)
{
```

(continues on next page)

(continued from previous page)

```
    musb->isr = jz4740_musb_interrupt;

    return 0;
}
```

This instruction sets a pointer to the glue layer IRQ handler function, in order for the controller hardware to call the handler back when an IRQ comes from the controller hardware. The interrupt handler is now implemented and registered.

20.14.4 Device Platform Data

In order to write an MUSB glue layer, you need to have some data describing the hardware capabilities of your controller hardware, which is called the platform data.

Platform data is specific to your hardware, though it may cover a broad range of devices, and is generally found somewhere in the `arch/` directory, depending on your device architecture.

For instance, platform data for the JZ4740 SoC is found in `arch/mips/jz4740/platform.c`. In the `platform.c` file each device of the JZ4740 SoC is described through a set of structures.

Here is the part of `arch/mips/jz4740/platform.c` that covers the USB Device Controller (UDC):

```
/* USB Device Controller */
struct platform_device jz4740_udc_xceiv_device = {
    .name = "usb_phy_gen_xceiv",
    .id   = 0,
};

static struct resource jz4740_udc_resources[] = {
    [0] = {
        .start = JZ4740_UDC_BASE_ADDR,
        .end   = JZ4740_UDC_BASE_ADDR + 0x10000 - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = JZ4740_IRQ_UDC,
        .end   = JZ4740_IRQ_UDC,
        .flags = IORESOURCE_IRQ,
        .name  = "mc",
    },
};

struct platform_device jz4740_udc_device = {
    .name = "musb-jz4740",
    .id   = -1,
    .dev  = {
        .dma_mask          = &jz4740_udc_device.dev.coherent_dma_
↪mask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    },
```

(continues on next page)

(continued from previous page)

```
    .num_resources = ARRAY_SIZE(jz4740_udc_resources),
    .resource      = jz4740_udc_resources,
};
```

The `jz4740_udc_xceiv_device` platform device structure (line 2) describes the UDC transceiver with a name and id number.

At the time of this writing, note that `usb_phy_gen_xceiv` is the specific name to be used for all transceivers that are either built-in with reference USB IP or autonomous and doesn't require any PHY programming. You will need to set `CONFIG_NOP_USB_XCEIV=y` in the kernel configuration to make use of the corresponding transceiver driver. The id field could be set to -1 (equivalent to `PLATFORM_DEVID_NONE`), -2 (equivalent to `PLATFORM_DEVID_AUTO`) or start with 0 for the first device of this kind if we want a specific id number.

The `jz4740_udc_resources` resource structure (line 7) defines the UDC registers base addresses.

The first array (line 9 to 11) defines the UDC registers base memory addresses: start points to the first register memory address, end points to the last register memory address and the flags member defines the type of resource we are dealing with. So `IORESOURCE_MEM` is used to define the registers memory addresses. The second array (line 14 to 17) defines the UDC IRQ registers addresses. Since there is only one IRQ register available for the JZ4740 UDC, start and end point at the same address. The `IORESOURCE_IRQ` flag tells that we are dealing with IRQ resources, and the name `mc` is in fact hard-coded in the MUSB core in order for the controller driver to retrieve this IRQ resource by querying it by its name.

Finally, the `jz4740_udc_device` platform device structure (line 21) describes the UDC itself.

The `musb-jz4740` name (line 22) defines the MUSB driver that is used for this device; remember this is in fact the name that we used in the `jz4740_driver` platform driver structure in Linux MUSB Basics. The id field (line 23) is set to -1 (equivalent to `PLATFORM_DEVID_NONE`) since we do not need an id for the device: the MUSB controller driver was already set to allocate an automatic id in Linux MUSB Basics. In the dev field we care for DMA related information here. The `dma_mask` field (line 25) defines the width of the DMA mask that is going to be used, and `coherent_dma_mask` (line 26) has the same purpose but for the `alloc_coherent` DMA mappings: in both cases we are using a 32 bits mask. Then the resource field (line 29) is simply a pointer to the resource structure defined before, while the `num_resources` field (line 28) keeps track of the number of arrays defined in the resource structure (in this case there were two resource arrays defined before).

With this quick overview of the UDC platform data at the `arch/` level now done, let's get back to the MUSB glue layer specific platform data in `drivers/usb/musb/jz4740.c`:

```
static struct musb_hdrc_config jz4740_musb_config = {
    /* Silicon does not implement USB OTG. */
    .multipoint = 0,
    /* Max EPs scanned, driver will decide which EP can be used.␣
↪*/
```

(continues on next page)

(continued from previous page)

```
    .num_eps    = 4,
    /* RAMbits needed to configure EPs from table */
    .ram_bits   = 9,
    .fifo_cfg = jz4740_musb_fifo_cfg,
    .fifo_cfg_size = ARRAY_SIZE(jz4740_musb_fifo_cfg),
};

static struct musb_hdrc_platform_data jz4740_musb_platform_data =
↪{
    .mode   = MUSB_PERIPHERAL,
    .config = &jz4740_musb_config,
};
```

First the glue layer configures some aspects of the controller driver operation related to the controller hardware specifics. This is done through the `jz4740_musb_config` `musb_hdrc_config` structure.

Defining the OTG capability of the controller hardware, the multipoint member (line 3) is set to 0 (equivalent to false) since the JZ4740 UDC is not OTG compatible. Then `num_eps` (line 5) defines the number of USB endpoints of the controller hardware, including endpoint 0: here we have 3 endpoints + endpoint 0. Next is `ram_bits` (line 7) which is the width of the RAM address bus for the MUSB controller hardware. This information is needed when the controller driver cannot automatically configure endpoints by reading the relevant controller hardware registers. This issue will be discussed when we get to device quirks in Device Quirks. Last two fields (line 8 and 9) are also about device quirks: `fifo_cfg` points to the USB endpoints configuration table and `fifo_cfg_size` keeps track of the size of the number of entries in that configuration table. More on that later in Device Quirks.

Then this configuration is embedded inside `jz4740_musb_platform_data` `musb_hdrc_platform_data` structure (line 11): config is a pointer to the configuration structure itself, and mode tells the controller driver if the controller hardware may be used as `MUSB_HOST` only, `MUSB_PERIPHERAL` only or `MUSB_OTG` which is a dual mode.

Remember that `jz4740_musb_platform_data` is then used to convey platform data information as we have seen in the probe function in Linux MUSB Basics.

20.14.5 Device Quirks

Completing the platform data specific to your device, you may also need to write some code in the glue layer to work around some device specific limitations. These quirks may be due to some hardware bugs, or simply be the result of an incomplete implementation of the USB On-the-Go specification.

The JZ4740 UDC exhibits such quirks, some of which we will discuss here for the sake of insight even though these might not be found in the controller hardware you are working on.

Let's get back to the init function first:

```
static int jz4740_musb_init(struct musb *musb)
{
    musb->xceiv = usb_get_phy(USB_PHY_TYPE_USB2);
    if (!musb->xceiv) {
        pr_err("HS UDC: no transceiver configured\n");
        return -ENODEV;
    }

    /* Silicon does not implement ConfigData register.
     * Set dyn_fifo to avoid reading EP config from hardware.
     */
    musb->dyn_fifo = true;

    musb->isr = jz4740_musb_interrupt;

    return 0;
}
```

Instruction on line 12 helps the MUSB controller driver to work around the fact that the controller hardware is missing registers that are used for USB endpoints configuration.

Without these registers, the controller driver is unable to read the endpoints configuration from the hardware, so we use line 12 instruction to bypass reading the configuration from silicon, and rely on a hard-coded table that describes the endpoints configuration instead:

```
static struct musb_fifo_cfg jz4740_musb_fifo_cfg[] = {
    { .hw_ep_num = 1, .style = FIFO_TX, .maxpacket = 512, },
    { .hw_ep_num = 1, .style = FIFO_RX, .maxpacket = 512, },
    { .hw_ep_num = 2, .style = FIFO_TX, .maxpacket = 64, },
};
```

Looking at the configuration table above, we see that each endpoints is described by three fields: `hw_ep_num` is the endpoint number, style is its direction (either `FIFO_TX` for the controller driver to send packets in the controller hardware, or `FIFO_RX` to receive packets from hardware), and maxpacket defines the maximum size of each data packet that can be transmitted over that endpoint. Reading from the table, the controller driver knows that endpoint 1 can be used to send and receive USB data packets of 512 bytes at once (this is in fact a bulk in/out endpoint), and endpoint 2 can be used to send data packets of 64 bytes at once (this is in fact an interrupt endpoint).

Note that there is no information about endpoint 0 here: that one is implemented by default in every silicon design, with a predefined configuration according to the USB specification. For more examples of endpoint configuration tables, see `musb_core.c`.

Let's now get back to the interrupt handler function:

```
static irqreturn_t jz4740_musb_interrupt(int irq, void *__hci)
{
    unsigned long   flags;
    irqreturn_t     retval = IRQ_NONE;
    struct musb     *musb = __hci;
```

(continues on next page)

(continued from previous page)

```
    spin_lock_irqsave(&musb->lock, flags);

    musb->int_usb = musb_readb(musb->mregs, MUSB_INTRUSB);
    musb->int_tx = musb_readw(musb->mregs, MUSB_INTRTX);
    musb->int_rx = musb_readw(musb->mregs, MUSB_INTRRX);

    /*
     * The controller is gadget only, the state of the host mode␣
↪IRQ bits is
     * undefined. Mask them to make sure that the musb driver␣
↪core will
     * never see them set
     */
    musb->int_usb &= MUSB_INTR_SUSPEND | MUSB_INTR_RESUME |
        MUSB_INTR_RESET | MUSB_INTR_SOF;

    if (musb->int_usb || musb->int_tx || musb->int_rx)
        retval = musb_interrupt(musb);

    spin_unlock_irqrestore(&musb->lock, flags);

    return retval;
}
```

Instruction on line 18 above is a way for the controller driver to work around the fact that some interrupt bits used for USB host mode operation are missing in the `MUSB_INTRUSB` register, thus left in an undefined hardware state, since this MUSB controller hardware is used in peripheral mode only. As a consequence, the glue layer masks these missing bits out to avoid parasite interrupts by doing a logical AND operation between the value read from `MUSB_INTRUSB` and the bits that are actually implemented in the register.

These are only a couple of the quirks found in the JZ4740 USB device controller. Some others were directly addressed in the MUSB core since the fixes were generic enough to provide a better handling of the issues for others controller hardware eventually.

20.14.6 Conclusion

Writing a Linux MUSB glue layer should be a more accessible task, as this documentation tries to show the ins and outs of this exercise.

The JZ4740 USB device controller being fairly simple, I hope its glue layer serves as a good example for the curious mind. Used with the current MUSB glue layers, this documentation should provide enough guidance to get started; should anything gets out of hand, the linux-usb mailing list archive is another helpful resource to browse through.

20.14.7 Acknowledgements

Many thanks to Lars-Peter Clausen and Maarten ter Huurne for answering my questions while I was writing the JZ4740 glue layer and for helping me out getting the code in good shape.

I would also like to thank the Qi-Hardware community at large for its cheerful guidance and support.

20.14.8 Resources

USB Home Page: https://www.usb.org

linux-usb Mailing List Archives: https://marc.info/?l=linux-usb

USB On-the-Go Basics: https://www.maximintegrated.com/app-notes/index.mvp/id/1822

Writing USB Device Drivers

Texas Instruments USB Configuration Wiki Page: http://processors.wiki.ti.com/index.php/Usbgeneralpage

20.15 USB Type-C connector class

20.15.1 Introduction

The typec class is meant for describing the USB Type-C ports in a system to the user space in unified fashion. The class is designed to provide nothing else except the user space interface implementation in hope that it can be utilized on as many platforms as possible.

The platforms are expected to register every USB Type-C port they have with the class. In a normal case the registration will be done by a USB Type-C or PD PHY driver, but it may be a driver for firmware interface such as UCSI, driver for USB PD controller or even driver for Thunderbolt3 controller. This document considers the component registering the USB Type-C ports with the class as “port driver”.

On top of showing the capabilities, the class also offer user space control over the roles and alternate modes of ports, partners and cable plugs when the port driver is capable of supporting those features.

The class provides an API for the port drivers described in this document. The attributes are described in Documentation/ABI/testing/sysfs-class-typec.

20.15.2 User space interface

Every port will be presented as its own device under /sys/class/typec/. The first port will be named “port0”, the second “port1” and so on.

When connected, the partner will be presented also as its own device under /sys/class/typec/. The parent of the partner device will always be the port it is attached to. The partner attached to port “port0” will be named “port0-partner”. Full path to the device would be /sys/class/typec/port0/port0-partner/.

The cable and the two plugs on it may also be optionally presented as their own devices under /sys/class/typec/. The cable attached to the port “port0” port will be named port0-cable and the plug on the SOP Prime end (see USB Power Delivery Specification ch. 2.4) will be named “port0-plug0” and on the SOP Double Prime end “port0-plug1”. The parent of a cable will always be the port, and the parent of the cable plugs will always be the cable.

If the port, partner or cable plug supports Alternate Modes, every supported Alternate Mode SVID will have their own device describing them. Note that the Alternate Mode devices will not be attached to the typec class. The parent of an alternate mode will be the device that supports it, so for example an alternate mode of port0-partner will be presented under /sys/class/typec/port0-partner/. Every mode that is supported will have its own group under the Alternate Mode device named “mode<index>”, for example /sys/class/typec/port0/<alternate mode>/mode1/. The requests for entering/exiting a mode can be done with “active” attribute file in that group.

20.15.3 Driver API

Registering the ports

The port drivers will describe every Type-C port they control with struct typec_capability data structure, and register them with the following API:

struct typec_port * **typec_register_port**(struct device * parent, const struct typec_capability * cap)
Register a USB Type-C Port

Parameters

struct device * parent Parent device

const struct typec_capability * cap Description of the port

Description

Registers a device for USB Type-C Port described in **cap**.

Returns handle to the port on success or ERR_PTR on failure.

void **typec_unregister_port**(struct typec_port * port)
Unregister a USB Type-C Port

Parameters

struct typec_port * port The port to be unregistered

Description

Unregister device created with `typec_register_port()`.

When registering the ports, the prefer_role member in struct typec_capability deserves special notice. If the port that is being registered does not have initial role preference, which means the port does not execute Try.SNK or Try.SRC by default, the member must have value TYPEC_NO_PREFERRED_ROLE. Otherwise if the port executes Try.SNK by default, the member must have value TYPEC_DEVICE, and with Try.SRC the value must be TYPEC_HOST.

Registering Partners

After successful connection of a partner, the port driver needs to register the partner with the class. Details about the partner need to be described in struct typec_partner_desc. The class copies the details of the partner during registration. The class offers the following API for registering/unregistering partners.

struct typec_partner * **typec_register_partner**(struct typec_port * port, struct typec_partner_desc * desc)
Register a USB Type-C Partner

Parameters

struct typec_port * port The USB Type-C Port the partner is connected to

struct typec_partner_desc * desc Description of the partner

Description

Registers a device for USB Type-C Partner described in **desc**.

Returns handle to the partner on success or ERR_PTR on failure.

void **typec_unregister_partner**(struct typec_partner * partner)
Unregister a USB Type-C Partner

Parameters

struct typec_partner * partner The partner to be unregistered

Description

Unregister device created with `typec_register_partner()`.

The class will provide a handle to struct typec_partner if the registration was successful, or NULL.

If the partner is USB Power Delivery capable, and the port driver is able to show the result of Discover Identity command, the partner descriptor structure should include handle to struct usb_pd_identity instance. The class will then create a sysfs directory for the identity under the partner device. The result of Discover Identity command can then be reported with the following API:

int **typec_partner_set_identity**(struct typec_partner * partner)
Report result from Discover Identity command

Parameters

struct typec_partner * partner The partner updated identity values

Description

This routine is used to report that the result of Discover Identity USB power delivery command has become available.

Registering Cables

After successful connection of a cable that supports USB Power Delivery Structured VDM "Discover Identity", the port driver needs to register the cable and one or two plugs, depending if there is CC Double Prime controller present in the cable or not. So a cable capable of SOP Prime communication, but not SOP Double Prime communication, should only have one plug registered. For more information about SOP communication, please read chapter about it from the latest USB Power Delivery specification.

The plugs are represented as their own devices. The cable is registered first, followed by registration of the cable plugs. The cable will be the parent device for the plugs. Details about the cable need to be described in struct typec_cable_desc and about a plug in struct typec_plug_desc. The class copies the details during registration. The class offers the following API for registering/unregistering cables and their plugs:

struct typec_plug * **typec_register_plug**(struct typec_cable * cable, struct typec_plug_desc * desc)
Register a USB Type-C Cable Plug

Parameters

struct typec_cable * cable USB Type-C Cable with the plug

struct typec_plug_desc * desc Description of the cable plug

Description

Registers a device for USB Type-C Cable Plug described in **desc**. A USB Type-C Cable Plug represents a plug with electronics in it that can response to USB Power Delivery SOP Prime or SOP Double Prime packages.

Returns handle to the cable plug on success or ERR_PTR on failure.

void **typec_unregister_plug**(struct typec_plug * plug)
Unregister a USB Type-C Cable Plug

Parameters

struct typec_plug * plug The cable plug to be unregistered

Description

Unregister device created with `typec_register_plug()`.

struct typec_cable * **typec_register_cable**(struct typec_port * port, struct typec_cable_desc * desc)
Register a USB Type-C Cable

Parameters

struct typec_port * port The USB Type-C Port the cable is connected to

struct typec_cable_desc * desc Description of the cable

Description

Registers a device for USB Type-C Cable described in **desc**. The cable will be parent for the optional cable plug devises.

Returns handle to the cable on success or ERR_PTR on failure.

void **typec_unregister_cable**(struct typec_cable * cable)
Unregister a USB Type-C Cable

Parameters

struct typec_cable * cable The cable to be unregistered

Description

Unregister device created with `typec_register_cable()`.

The class will provide a handle to struct typec_cable and struct typec_plug if the registration is successful, or NULL if it isn't.

If the cable is USB Power Delivery capable, and the port driver is able to show the result of Discover Identity command, the cable descriptor structure should include handle to struct usb_pd_identity instance. The class will then create a sysfs directory for the identity under the cable device. The result of Discover Identity command can then be reported with the following API:

int **typec_cable_set_identity**(struct typec_cable * cable)
Report result from Discover Identity command

Parameters

struct typec_cable * cable The cable updated identity values

Description

This routine is used to report that the result of Discover Identity USB power delivery command has become available.

Notifications

When the partner has executed a role change, or when the default roles change during connection of a partner or cable, the port driver must use the following APIs to report it to the class:

void **typec_set_data_role**(struct typec_port * port, enum typec_data_role role)
Report data role change

Parameters

struct typec_port * port The USB Type-C Port where the role was changed

enum typec_data_role role The new data role

Description

This routine is used by the port drivers to report data role changes.

void **typec_set_pwr_role**(struct typec_port * port, enum typec_role role)
Report power role change

Parameters

struct typec_port * port The USB Type-C Port where the role was changed

enum typec_role role The new data role

Description

This routine is used by the port drivers to report power role changes.

void **typec_set_vconn_role**(struct typec_port * port, enum typec_role role)
Report VCONN source change

Parameters

struct typec_port * port The USB Type-C Port which VCONN role changed

enum typec_role role Source when **port** is sourcing VCONN, or Sink when it's not

Description

This routine is used by the port drivers to report if the VCONN source is changes.

void **typec_set_pwr_opmode**(struct typec_port * port, enum typec_pwr_opmode opmode)
Report changed power operation mode

Parameters

struct typec_port * port The USB Type-C Port where the mode was changed

enum typec_pwr_opmode opmode New power operation mode

Description

This routine is used by the port drivers to report changed power operation mode in **port**. The modes are USB (default), 1.5A, 3.0A as defined in USB Type-C specification, and "USB Power Delivery" when the power levels are negotiated with methods defined in USB Power Delivery specification.

Alternate Modes

USB Type-C ports, partners and cable plugs may support Alternate Modes. Each Alternate Mode will have identifier called SVID, which is either a Standard ID given by USB-IF or vendor ID, and each supported SVID can have 1 - 6 modes. The class provides struct typec_mode_desc for describing individual mode of a SVID, and struct typec_altmode_desc which is a container for all the supported modes.

Ports that support Alternate Modes need to register each SVID they support with the following API:

struct typec_altmode * **typec_port_register_altmode**(struct typec_port * port, const struct typec_altmode_desc * desc)

Register USB Type-C Port Alternate Mode

Parameters

struct typec_port * port USB Type-C Port that supports the alternate mode

const struct typec_altmode_desc * desc Description of the alternate mode

Description

This routine is used to register an alternate mode that **port** is capable of supporting.

Returns handle to the alternate mode on success or ERR_PTR on failure.

If a partner or cable plug provides a list of SVIDs as response to USB Power Delivery Structured VDM Discover SVIDs message, each SVID needs to be registered.

API for the partners:

struct typec_altmode * **typec_partner_register_altmode**(struct typec_partner * partner, const struct typec_altmode_desc * desc)

Register USB Type-C Partner Alternate Mode

Parameters

struct typec_partner * partner USB Type-C Partner that supports the alternate mode

const struct typec_altmode_desc * desc Description of the alternate mode

Description

This routine is used to register each alternate mode individually that **partner** has listed in response to Discover SVIDs command. The modes for a SVID listed in response to Discover Modes command need to be listed in an array in **desc**.

Returns handle to the alternate mode on success or NULL on failure.

API for the Cable Plugs:

struct typec_altmode * **typec_plug_register_altmode**(struct typec_plug * plug, const struct typec_altmode_desc * desc)

Register USB Type-C Cable Plug Alternate Mode

Parameters

struct typec_plug * plug USB Type-C Cable Plug that supports the alternate mode

const struct typec_altmode_desc * desc Description of the alternate mode

Description

This routine is used to register each alternate mode individually that **plug** has listed in response to Discover SVIDs command. The modes for a SVID that the

plug lists in response to Discover Modes command need to be listed in an array in **desc**.

Returns handle to the alternate mode on success or ERR_PTR on failure.

So ports, partners and cable plugs will register the alternate modes with their own functions, but the registration will always return a handle to struct typec_altmode on success, or NULL. The unregistration will happen with the same function:

void **typec_unregister_altmode**(struct typec_altmode * adev)
Unregister Alternate Mode

Parameters

struct typec_altmode * adev The alternate mode to be unregistered

Description

Unregister device created with `typec_partner_register_altmode()`, `typec_plug_register_altmode()` or `typec_port_register_altmode()`.

If a partner or cable plug enters or exits a mode, the port driver needs to notify the class with the following API:

void **typec_altmode_update_active**(struct typec_altmode * adev, bool active)
Report Enter/Exit mode

Parameters

struct typec_altmode * adev Handle to the alternate mode

bool active True when the mode has been entered

Description

If a partner or cable plug executes Enter/Exit Mode command successfully, the drivers use this routine to report the updated state of the mode.

Multiplexer/DeMultiplexer Switches

USB Type-C connectors may have one or more mux/demux switches behind them. Since the plugs can be inserted right-side-up or upside-down, a switch is needed to route the correct data pairs from the connector to the USB controllers. If Alternate or Accessory Modes are supported, another switch is needed that can route the pins on the connector to some other component besides USB. USB Type-C Connector Class supplies an API for registering those switches.

struct typec_switch * **typec_switch_register**(struct device * parent, const struct typec_switch_desc * desc)
Register USB Type-C orientation switch

Parameters

struct device * parent Parent device

const struct typec_switch_desc * desc Orientation switch description

Description

This function registers a switch that can be used for routing the correct data pairs depending on the cable plug orientation from the USB Type-C connector to the USB controllers. USB Type-C plugs can be inserted right-side-up or upside-down.

void **typec_switch_unregister**(struct typec_switch * sw)
Unregister USB Type-C orientation switch

Parameters

struct typec_switch * sw USB Type-C orientation switch

Description

Unregister switch that was registered with `typec_switch_register()`.

struct typec_mux * **typec_mux_register**(struct device * parent, const struct typec_mux_desc * desc)
Register Multiplexer routing USB Type-C pins

Parameters

struct device * parent Parent device

const struct typec_mux_desc * desc Multiplexer description

Description

USB Type-C connectors can be used for alternate modes of operation besides USB when Accessory/Alternate Modes are supported. With some of those modes, the pins on the connector need to be reconfigured. This function registers multiplexer switches routing the pins on the connector.

void **typec_mux_unregister**(struct typec_mux * mux)
Unregister Multiplexer Switch

Parameters

struct typec_mux * mux USB Type-C Connector Multiplexer/DeMultiplexer

Description

Unregister mux that was registered with `typec_mux_register()`.

In most cases the same physical mux will handle both the orientation and mode. However, as the port drivers will be responsible for the orientation, and the alternate mode drivers for the mode, the two are always separated into their own logical components: “mux” for the mode and “switch” for the orientation.

When a port is registered, USB Type-C Connector Class requests both the mux and the switch for the port. The drivers can then use the following API for controlling them:

int **typec_set_orientation**(struct typec_port * port, enum typec_orientation orientation)
Set USB Type-C cable plug orientation

Parameters

struct typec_port * port USB Type-C Port

enum typec_orientation orientation USB Type-C cable plug orientation

Description

Set cable plug orientation for **port**.

int **typec_set_mode**(struct typec_port * port, int mode)
Set mode of operation for USB Type-C connector

Parameters

struct typec_port * port USB Type-C connector

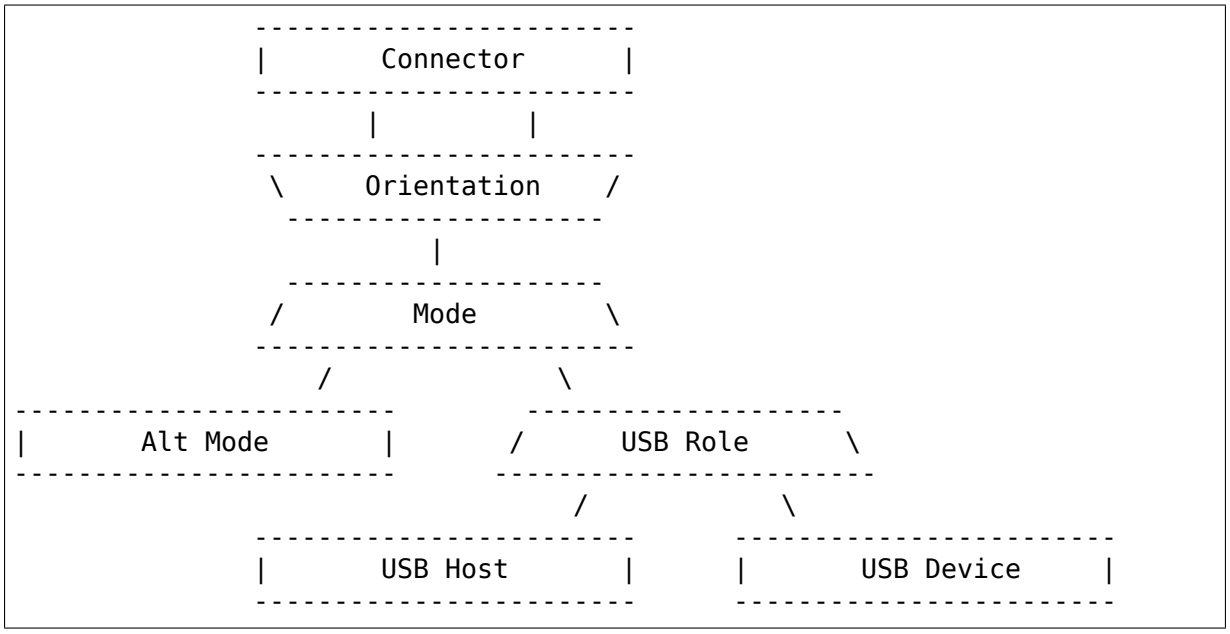
int mode Accessory Mode, USB Operation or Safe State

Description

Configure **port** for Accessory Mode **mode**. This function will configure the muxes needed for **mode**.

If the connector is dual-role capable, there may also be a switch for the data role. USB Type-C Connector Class does not supply separate API for them. The port drivers can use USB Role Class API with those.

Illustration of the muxes behind a connector that supports an alternate mode:



20.16 API for USB Type-C Alternate Mode drivers

20.16.1 Introduction

Alternate modes require communication with the partner using Vendor Defined Messages (VDM) as defined in USB Type-C and USB Power Delivery Specifications. The communication is SVID (Standard or Vendor ID) specific, i.e. specific for every alternate mode, so every alternate mode will need a custom driver.

USB Type-C bus allows binding a driver to the discovered partner alternate modes by using the SVID and the mode number.

USB Type-C Connector Class provides a device for every alternate mode a port supports, and separate device for every alternate mode the partner supports. The drivers for the alternate modes are bound to the partner alternate mode devices, and the port alternate mode devices must be handled by the port drivers.

When a new partner alternate mode device is registered, it is linked to the alternate mode device of the port that the partner is attached to, that has matching SVID and mode. Communication between the port driver and alternate mode driver will happen using the same API.

The port alternate mode devices are used as a proxy between the partner and the alternate mode drivers, so the port drivers are only expected to pass the SVID specific commands from the alternate mode drivers to the partner, and from the partners to the alternate mode drivers. No direct SVID specific communication is needed from the port drivers, but the port drivers need to provide the operation callbacks for the port alternate mode devices, just like the alternate mode drivers need to provide them for the partner alternate mode devices.

20.16.2 Usage:

General

By default, the alternate mode drivers are responsible for entering the mode. It is also possible to leave the decision about entering the mode to the user space (See Documentation/ABI/testing/sysfs-class-typec). Port drivers should not enter any modes on their own.

`->vdm` is the most important callback in the operation callbacks vector. It will be used to deliver all the SVID specific commands from the partner to the alternate mode driver, and vice versa in case of port drivers. The drivers send the SVID specific commands to each other using `typec_altmode_vdm()`.

If the communication with the partner using the SVID specific commands results in need to reconfigure the pins on the connector, the alternate mode driver needs to notify the bus using `typec_altmode_notify()`. The driver passes the negotiated SVID specific pin configuration value to the function as parameter. The bus driver will then configure the mux behind the connector using that value as the state value for the mux.

NOTE: The SVID specific pin configuration values must always start from `TYPEC_STATE_MODAL`. USB Type-C specification defines two default states for the connector: `TYPEC_STATE_USB` and `TYPEC_STATE_SAFE`. These values are reserved by the bus as the first possible values for the state. When the alternate mode is entered, the bus will put the connector into `TYPEC_STATE_SAFE` before sending Enter or Exit Mode command as defined in USB Type-C Specification, and also put the connector back to `TYPEC_STATE_USB` after the mode has been exited.

An example of working definitions for SVID specific pin configurations would look like this:

```
enum {
    ALTMODEX_CONF_A = TYPEC_STATE_MODAL,
    ALTMODEX_CONF_B,
```

(continues on next page)

(continued from previous page)

```
    ...
};
```

Helper macro `TYPEC_MODAL_STATE()` can also be used:

```
#define ALTMODEX_CONF_A = TYPEC_MODAL_STATE(0);
#define ALTMODEX_CONF_B = TYPEC_MODAL_STATE(1);
```

Cable plug alternate modes

The alternate mode drivers are not bound to cable plug alternate mode devices, only to the partner alternate mode devices. If the alternate mode supports, or requires, a cable that responds to SOP Prime, and optionally SOP Double Prime messages, the driver for that alternate mode must request handle to the cable plug alternate modes using `typec_altmode_get_plug()`, and take over their control.

20.16.3 Driver API

Alternate mode driver registering/unregistering

Alternate mode driver operations

int **typec_altmode_notify**(struct typec_altmode * adev, unsigned long conf, void * data)
Communication between the OS and alternate mode driver

Parameters

struct typec_altmode * adev Handle to the alternate mode

unsigned long conf Alternate mode specific configuration value

void * data Alternate mode specific data

Description

The primary purpose for this function is to allow the alternate mode drivers to tell which pin configuration has been negotiated with the partner. That information will then be used for example to configure the muxes. Communication to the other direction is also possible, and low level device drivers can also send notifications to the alternate mode drivers. The actual communication will be specific for every SVID.

int **typec_altmode_enter**(struct typec_altmode * adev, u32 * vdo)
Enter Mode

Parameters

struct typec_altmode * adev The alternate mode

u32 * vdo VDO for the Enter Mode command

Description

The alternate mode drivers use this function to enter mode. The port drivers use this to inform the alternate mode drivers that the partner has initiated Enter Mode command. If the alternate mode does not require VDO, **vdo** must be NULL.

int **typec_altmode_exit**(struct typec_altmode * adev)
Exit Mode

Parameters

struct typec_altmode * adev The alternate mode

Description

The partner of **adev** has initiated Exit Mode command.

void **typec_altmode_attention**(struct typec_altmode * adev, u32 vdo)
Attention command

Parameters

struct typec_altmode * adev The alternate mode

u32 vdo VDO for the Attention command

Description

Notifies the partner of **adev** about Attention command.

int **typec_altmode_vdm**(struct typec_altmode * adev, const u32 header, const u32 * vdo, int count)
Send Vendor Defined Messages (VDM) to the partner

Parameters

struct typec_altmode * adev Alternate mode handle

const u32 header VDM Header

const u32 * vdo Array of Vendor Defined Data Objects

int count Number of Data Objects

Description

The alternate mode drivers use this function for SVID specific communication with the partner. The port drivers use it to deliver the Structured VDMs received from the partners to the alternate mode drivers.

API for the port drivers

struct typec_altmode * **typec_match_altmode**(struct typec_altmode ** altmodes, size_t n, u16 svid, u8 mode)
Match SVID and mode to an array of alternate modes

Parameters

struct typec_altmode ** altmodes Array of alternate modes

size_t n Number of elements in the array, or -1 for NULL terminated arrays

u16 svid Standard or Vendor ID to match with

u8 mode Mode to match with

Description

Return pointer to an alternate mode with SVID matching **svid**, or NULL when no match is found.

Cable Plug operations

struct typec_altmode * **typec_altmode_get_plug**(struct typec_altmode * adev, enum typec_plug_index index)
Find cable plug alternate mode

Parameters

struct typec_altmode * adev Handle to partner alternate mode

enum typec_plug_index index Cable plug index

Description

Increment reference count for cable plug alternate mode device. Returns handle to the cable plug alternate mode, or NULL if none is found.

void **typec_altmode_put_plug**(struct typec_altmode * plug)
Decrement cable plug alternate mode reference count

Parameters

struct typec_altmode * plug Handle to the cable plug alternate mode

20.17 USB3 debug port

Author Lu Baolu <baolu.lu@linux.intel.com>

Date March 2017

20.17.1 GENERAL

This is a HOWTO for using the USB3 debug port on x86 systems.

Before using any kernel debugging functionality based on USB3 debug port, you need to:

```
1) check whether any USB3 debug port is available in
   your system;
2) check which port is used for debugging purposes;
3) have a USB 3.0 super-speed A-to-A debugging cable.
```

20.17.2 INTRODUCTION

The xHCI debug capability (DbC) is an optional but standalone functionality provided by the xHCI host controller. The xHCI specification describes DbC in the section 7.6.

When DbC is initialized and enabled, it will present a debug device through the debug port (normally the first USB3 super-speed port). The debug device is fully compliant with the USB framework and provides the equivalent of a very high performance full-duplex serial link between the debug target (the system under debugging) and a debug host.

20.17.3 EARLY PRINTK

DbC has been designed to log early printk messages. One use for this feature is kernel debugging. For example, when your machine crashes very early before the regular console code is initialized. Other uses include simpler, lockless logging instead of a full- blown printk console driver and klogd.

On the debug target system, you need to customize a debugging kernel with CONFIG_EARLY_PRINTK_USB_XDBC enabled. And, add below kernel boot parameter:

```
"earlyprintk=xdbc"
```

If there are multiple xHCI controllers in your system, you can append a host contoller index to this kernel parameter. This index starts from 0.

Current design doesn’ t support DbC runtime suspend/resume. As the result, you’ d better disable runtime power management for USB subsystem by adding below kernel boot parameter:

```
"usbcore.autosuspend=-1"
```

Before starting the debug target, you should connect the debug port to a USB port (root port or port of any external hub) on the debug host. The cable used to connect these two ports should be a USB 3.0 super-speed A-to-A debugging cable.

During early boot of the debug target, DbC will be detected and initialized. After initialization, the debug host should be able to enumerate the debug device in debug target. The debug host will then bind the debug device with the usb_debug driver module and create the /dev/ttyUSB device.

If the debug device enumeration goes smoothly, you should be able to see below kernel messages on the debug host:

```
# tail -f /var/log/kern.log
[ 1815.983374] usb 4-3: new SuperSpeed USB device number 4 using xhci_hcd
[ 1815.999595] usb 4-3: LPM exit latency is zeroed, disabling LPM.
[ 1815.999899] usb 4-3: New USB device found, idVendor=1d6b, idProduct=0004
[ 1815.999902] usb 4-3: New USB device strings: Mfr=1, Product=2,␣
↪SerialNumber=3
[ 1815.999903] usb 4-3: Product: Remote GDB
[ 1815.999904] usb 4-3: Manufacturer: Linux
[ 1815.999905] usb 4-3: SerialNumber: 0001
```

(continues on next page)

(continued from previous page)

```
[ 1816.000240] usb_debug 4-3:1.0: xhci_dbc converter detected
[ 1816.000360] usb 4-3: xhci_dbc converter now attached to ttyUSB0
```

You can use any communication program, for example minicom, to read and view the messages. Below simple bash scripts can help you to check the sanity of the setup.

```
===== start of bash scripts =============
#!/bin/bash

while true ; do
        while [ ! -d /sys/class/tty/ttyUSB0 ] ; do
                :
        done
cat /dev/ttyUSB0
done
===== end of bash scripts ===============
```

20.17.4 Serial TTY

The DbC support has been added to the xHCI driver. You can get a debug device provided by the DbC at runtime.

In order to use this, you need to make sure your kernel has been configured to support USB_XHCI_DBGCAP. A sysfs attribute under the xHCI device node is used to enable or disable DbC. By default, DbC is disabled:

```
root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc
disabled
```

Enable DbC with the following command:

```
root@target:/sys/bus/pci/devices/0000:00:14.0# echo enable > dbc
```

You can check the DbC state at anytime:

```
root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc
enabled
```

Connect the debug target to the debug host with a USB 3.0 super- speed A-to-A debugging cable. You can see /dev/ttyDBC0 created on the debug target. You will see below kernel message lines:

```
root@target: tail -f /var/log/kern.log
[  182.730103] xhci_hcd 0000:00:14.0: DbC connected
[  191.169420] xhci_hcd 0000:00:14.0: DbC configured
[  191.169597] xhci_hcd 0000:00:14.0: DbC now attached to /dev/ttyDBC0
```

Accordingly, the DbC state has been brought up to:

```
root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc
configured
```

On the debug host, you will see the debug device has been enumerated. You will see below kernel message lines:

```
root@host: tail -f /var/log/kern.log
[   79.454780] usb 2-2.1: new SuperSpeed USB device number 3 using xhci_hcd
[   79.475003] usb 2-2.1: LPM exit latency is zeroed, disabling LPM.
[   79.475389] usb 2-2.1: New USB device found, idVendor=1d6b,␣
↪idProduct=0010
[   79.475390] usb 2-2.1: New USB device strings: Mfr=1, Product=2,␣
↪SerialNumber=3
[   79.475391] usb 2-2.1: Product: Linux USB Debug Target
[   79.475392] usb 2-2.1: Manufacturer: Linux Foundation
[   79.475393] usb 2-2.1: SerialNumber: 0001
```

The debug device works now. You can use any communication or debugging program to talk between the host and the target.

CHAPTER

TWENTYONE

FIREWIRE (IEEE 1394) DRIVER INTERFACE GUIDE

21.1 Introduction and Overview

The Linux FireWire subsystem adds some interfaces into the Linux system to use/maintain+any resource on IEEE 1394 bus.

The main purpose of these interfaces is to access address space on each node on IEEE 1394 bus by ISO/IEC 13213 (IEEE 1212) procedure, and to control isochronous resources on the bus by IEEE 1394 procedure.

Two types of interfaces are added, according to consumers of the interface. A set of userspace interfaces is available via firewire character devices. A set of kernel interfaces is available via exported symbols in firewire-core module.

21.2 Firewire char device data structures

```
What:           /dev/fw[0-9]+
Date:           May 2007
KernelVersion:  2.6.22
Contact:        linux1394-devel@lists.sourceforge.net
Description:
                The character device files /dev/fw* are the␣
↪interface between
                firewire-core and IEEE 1394 device drivers␣
↪implemented in
                userspace.  The ioctl(2)- and read(2)-based ABI is␣
↪defined and
                documented in <linux/firewire-cdev.h>.

                This ABI offers most of the features which␣
↪firewire-core also
                exposes to kernelspace IEEE 1394 drivers.

                Each /dev/fw* is associated with one IEEE 1394 node,
↪ which can
                be remote or local nodes.  Operations on a /dev/fw*␣
↪file have
                different scope:
                  - The 1394 node which is associated with the file:
```

```
                        - Asynchronous request transmission
                        - Get the Configuration ROM
                        - Query node ID
                        - Query maximum speed of the path between␣
↪this node
                          and local node
                  - The 1394 bus (i.e. "card") to which the node is␣
↪attached to:
                        - Isochronous stream transmission and␣
↪reception
                        - Asynchronous stream transmission and␣
↪reception
                        - Asynchronous broadcast request␣
↪transmission
                        - PHY packet transmission and reception
                        - Allocate, reallocate, deallocate␣
↪isochronous
                          resources (channels, bandwidth) at the␣
↪bus's IRM
                        - Query node IDs of local node, root node,
↪ IRM, bus
                          manager
                        - Query cycle time
                        - Bus reset initiation, bus reset event␣
↪reception
                  - All 1394 buses:
                        - Allocation of IEEE 1212 address ranges␣
↪on the local
                          link layers, reception of inbound␣
↪requests to such
                          an address range, asynchronous response␣
↪transmission
                          to inbound requests
                        - Addition of descriptors or directories␣
↪to the local
                          nodes' Configuration ROM

                Due to the different scope of operations and in␣
↪order to let
                userland implement different access permission␣
↪models, some
                operations are restricted to /dev/fw* files that␣
↪are associated
                with a local node:
                        - Addition of descriptors or directories␣
↪to the local
                          nodes' Configuration ROM
                        - PHY packet transmission and reception

                A /dev/fw* file remains associated with one␣
↪particular node
```

```
                during its entire life time.  Bus topology changes,␣
↪and hence
                node ID changes, are tracked by firewire-core.  ABI␣
↪users do not
                need to be aware of topology.

                The following file operations are supported:

                open(2)
                Currently the only useful flags are O_RDWR.

                ioctl(2)
                Initiate various actions.  Some take immediate␣
↪effect, others
                are performed asynchronously while or after the␣
↪ioctl returns.
                See the inline documentation in <linux/
↪firewire-cdev.h> for
                descriptions of all ioctls.

                poll(2), select(2), epoll_wait(2) etc.
                Watch for events to become available to be read.

                read(2)
                Receive various events.  There are solicited events␣
↪like
                outbound asynchronous transaction completion or␣
↪isochronous
                buffer completion, and unsolicited events such as␣
↪bus resets,
                request reception, or PHY packet reception.  Always␣
↪use a read
                buffer which is large enough to receive the largest␣
↪event that
                could ever arrive.  See <linux/firewire-cdev.h> for␣
↪descriptions
                of all event types and for which ioctls affect␣
↪reception of
                events.

                mmap(2)
                Allocate a DMA buffer for isochronous reception or␣
↪transmission
                and map it into the process address space.  The␣
↪arguments should
                be used as follows:  addr = NULL, length = the␣
↪desired buffer
                size, i.e. number of packets times size of largest␣
↪packet,
                prot = at least PROT_READ for reception and at␣
↪least PROT_WRITE
```

```
                for transmission, flags = MAP_SHARED, fd = the␣
↪handle to the
                /dev/fw*, offset = 0.

                Isochronous reception works in packet-per-buffer␣
↪fashion except
                for multichannel reception which works in␣
↪buffer-fill mode.

                munmap(2)
                Unmap the isochronous I/O buffer from the process␣
↪address space.

                close(2)
                Besides stopping and freeing I/O contexts that were␣
↪associated
                with the file descriptor, back out any changes to␣
↪the local
                nodes' Configuration ROM.  Deallocate isochronous␣
↪channels and
                bandwidth at the IRM that were marked for␣
↪kernel-assisted
                re- and deallocation.

Users:          libraw1394
                libdc1394
                libhinawa
                tools like linux-firewire-utils, fwhack, ...
```

struct **fw_cdev_event_common**
Common part of all fw_cdev_event_* types

Definition

```
struct fw_cdev_event_common {
  __u64 closure;
  __u32 type;
};
```

Members

closure For arbitrary use by userspace

type Discriminates the fw_cdev_event_* types

Description

This struct may be used to access generic members of all fw_cdev_event_* types regardless of the specific type.

Data passed in the **closure** field for a request will be returned in the corresponding event. It is big enough to hold a pointer on all platforms. The ioctl used to set **closure** depends on the **type** of event.

struct **fw_cdev_event_bus_reset**
Sent when a bus reset occurred

Definition

```
struct fw_cdev_event_bus_reset {
  __u64 closure;
  __u32 type;
  __u32 node_id;
  __u32 local_node_id;
  __u32 bm_node_id;
  __u32 irm_node_id;
  __u32 root_node_id;
  __u32 generation;
};
```

Members

closure See fw_cdev_event_common; set by FW_CDEV_IOC_GET_INFO ioctl

type See fw_cdev_event_common; always FW_CDEV_EVENT_BUS_RESET

node_id New node ID of this node

local_node_id Node ID of the local node, i.e. of the controller

bm_node_id Node ID of the bus manager

irm_node_id Node ID of the iso resource manager

root_node_id Node ID of the root node

generation New bus generation

Description

This event is sent when the bus the device belongs to goes through a bus reset. It provides information about the new bus configuration, such as new node ID for this device, new root ID, and others.

If **bm_node_id** is 0xffff right after bus reset it can be reread by an FW_CDEV_IOC_GET_INFO ioctl after bus manager selection was finished. Kernels with ABI version < 4 do not set **bm_node_id**.

struct **fw_cdev_event_response**
Sent when a response packet was received

Definition

```
struct fw_cdev_event_response {
  __u64 closure;
  __u32 type;
  __u32 rcode;
  __u32 length;
  __u32 data[0];
};
```

Members

closure See fw_cdev_event_common; set by FW_CDEV_IOC_SEND_REQUEST or FW_CDEV_IOC_SEND_BROADCAST_REQUEST or FW_CDEV_IOC_SEND_STREAM_PACKET ioctl

type See fw_cdev_event_common; always FW_CDEV_EVENT_RESPONSE

rcode Response code returned by the remote node

length Data length, i.e. the response's payload size in bytes

data Payload data, if any

Description

This event is sent when the stack receives a response to an outgoing request sent by FW_CDEV_IOC_SEND_REQUEST ioctl. The payload data for responses carrying data (read and lock responses) follows immediately and can be accessed through the **data** field.

The event is also generated after conclusions of transactions that do not involve response packets. This includes unified write transactions, broadcast write transactions, and transmission of asynchronous stream packets. **rcode** indicates success or failure of such transmissions.

struct **fw_cdev_event_request**
Old version of fw_cdev_event_request2

Definition

```
struct fw_cdev_event_request {
  __u64 closure;
  __u32 type;
  __u32 tcode;
  __u64 offset;
  __u32 handle;
  __u32 length;
  __u32 data[0];
};
```

Members

closure See fw_cdev_event_common; set by FW_CDEV_IOC_ALLOCATE ioctl

type See fw_cdev_event_common; always FW_CDEV_EVENT_REQUEST

tcode Transaction code of the incoming request

offset The offset into the 48-bit per-node address space

handle Reference to the kernel-side pending request

length Data length, i.e. the request's payload size in bytes

data Incoming data, if any

Description

This event is sent instead of fw_cdev_event_request2 if the kernel or the client implements ABI version <= 3. fw_cdev_event_request lacks essential information; use fw_cdev_event_request2 instead.

struct **fw_cdev_event_request2**
Sent on incoming request to an address region

Definition

```
struct fw_cdev_event_request2 {
  __u64 closure;
  __u32 type;
  __u32 tcode;
  __u64 offset;
  __u32 source_node_id;
  __u32 destination_node_id;
  __u32 card;
  __u32 generation;
  __u32 handle;
  __u32 length;
  __u32 data[0];
};
```

Members

closure See fw_cdev_event_common; set by FW_CDEV_IOC_ALLOCATE ioctl

type See fw_cdev_event_common; always FW_CDEV_EVENT_REQUEST2

tcode Transaction code of the incoming request

offset The offset into the 48-bit per-node address space

source_node_id Sender node ID

destination_node_id Destination node ID

card The index of the card from which the request came

generation Bus generation in which the request is valid

handle Reference to the kernel-side pending request

length Data length, i.e. the request’ s payload size in bytes

data Incoming data, if any

Description

This event is sent when the stack receives an incoming request to an address region registered using the FW_CDEV_IOC_ALLOCATE ioctl. The request is guaranteed to be completely contained in the specified region. Userspace is responsible for sending the response by FW_CDEV_IOC_SEND_RESPONSE ioctl, using the same **handle**.

The payload data for requests carrying data (write and lock requests) follows immediately and can be accessed through the **data** field.

Unlike fw_cdev_event_request, **tcode** of lock requests is one of the firewire-core specific TCODE_LOCK_MASK_SWAP…``TCODE_LOCK_VENDOR_DEPENDENT``, i.e. encodes the extended transaction code.

card may differ from fw_cdev_get_info.card because requests are received from all cards of the Linux host. **source_node_id**, **destination_node_id**, and **generation** pertain to that card. Destination node ID and bus generation may therefore differ from the corresponding fields of the last fw_cdev_event_bus_reset.

destination_node_id may also differ from the current node ID because of a non-local bus ID part or in case of a broadcast write request. Note, a client must call an FW_CDEV_IOC_SEND_RESPONSE ioctl even in case of a broadcast write request; the kernel will then release the kernel-side pending request but will not actually send a response packet.

In case of a write request to FCP_REQUEST or FCP_RESPONSE, the kernel already sent a write response immediately after the request was received; in this case the client must still call an FW_CDEV_IOC_SEND_RESPONSE ioctl to release the kernel-side pending request, though another response won't be sent.

If the client subsequently needs to initiate requests to the sender node of an fw_cdev_event_request2, it needs to use a device file with matching card index, node ID, and generation for outbound requests.

struct **fw_cdev_event_iso_interrupt**
Sent when an iso packet was completed

Definition

```
struct fw_cdev_event_iso_interrupt {
  __u64 closure;
  __u32 type;
  __u32 cycle;
  __u32 header_length;
  __u32 header[0];
};
```

Members

closure See fw_cdev_event_common; set by FW_CDEV_CREATE_ISO_CONTEXT ioctl

type See fw_cdev_event_common; always FW_CDEV_EVENT_ISO_INTERRUPT

cycle Cycle counter of the last completed packet

header_length Total length of following headers, in bytes

header Stripped headers, if any

Description

This event is sent when the controller has completed an fw_cdev_iso_packet with the FW_CDEV_ISO_INTERRUPT bit set, when explicitly requested with FW_CDEV_IOC_FLUSH_ISO, or when there have been so many completed packets without the interrupt bit set that the kernel's internal buffer for **header** is about to overflow. (In the last case, ABI versions < 5 drop header data up to the next interrupt packet.)

Isochronous transmit events (context type FW_CDEV_ISO_CONTEXT_TRANSMIT):

In version 3 and some implementations of version 2 of the ABI, header_length is a multiple of 4 and header contains timestamps of all packets up until the interrupt packet. The format of the timestamps is as described below for isochronous reception. In version 1 of the ABI, header_length was 0.

Isochronous receive events (context type FW_CDEV_ISO_CONTEXT_RECEIVE):

The headers stripped of all packets up until and including the interrupt packet are returned in the **header** field. The amount of header data per packet is as specified

at iso context creation by `fw_cdev_create_iso_context.header_size`.

Hence, _interrupt.header_length / _context.header_size is the number of packets received in this interrupt event. The client can now iterate through the mmap()'ed DMA buffer according to this number of packets and to the buffer sizes as the client specified in `fw_cdev_queue_iso`.

Since version 2 of this ABI, the portion for each packet in _interrupt.header consists of the 1394 isochronous packet header, followed by a timestamp quadlet if `fw_cdev_create_iso_context.header_size` > 4, followed by quadlets from the packet payload if `fw_cdev_create_iso_context.header_size` > 8.

Format of 1394 iso packet header: 16 bits data_length, 2 bits tag, 6 bits channel, 4 bits tcode, 4 bits sy, in big endian byte order. data_length is the actual received size of the packet without the four 1394 iso packet header bytes.

Format of timestamp: 16 bits invalid, 3 bits cycleSeconds, 13 bits cycleCount, in big endian byte order.

In version 1 of the ABI, no timestamp quadlet was inserted; instead, payload data followed directly after the 1394 is header if header_size > 4. Behaviour of ver. 1 of this ABI is no longer available since ABI ver. 2.

struct **fw_cdev_event_iso_interrupt_mc**
An iso buffer chunk was completed

Definition

```
struct fw_cdev_event_iso_interrupt_mc {
  __u64 closure;
  __u32 type;
  __u32 completed;
};
```

Members

`closure` See `fw_cdev_event_common`; set by `FW_CDEV_CREATE_ISO_CONTEXT` ioctl

`type` `FW_CDEV_EVENT_ISO_INTERRUPT_MULTICHANNEL`

`completed` Offset into the receive buffer; data before this offset is valid

Description

This event is sent in multichannel contexts (context type `FW_CDEV_ISO_CONTEXT_RECEIVE_MULTICHANNEL`) for `fw_cdev_iso_packet` buffer chunks that have been completely filled and that have the `FW_CDEV_ISO_INTERRUPT` bit set, or when explicitly requested with `FW_CDEV_IOC_FLUSH_ISO`.

The buffer is continuously filled with the following data, per packet:

- the 1394 iso packet header as described at `fw_cdev_event_iso_interrupt`, but in little endian byte order,
- packet payload (as many bytes as specified in the data_length field of the 1394 iso packet header) in big endian byte order,
- 0⋯3 padding bytes as needed to align the following trailer quadlet,
- trailer quadlet, containing the reception timestamp as described at `fw_cdev_event_iso_interrupt`, but in little endian byte order.

Hence the per-packet size is data_length (rounded up to a multiple of 4) + 8. When processing the data, stop before a packet that would cross the **completed** offset.

A packet near the end of a buffer chunk will typically spill over into the next queued buffer chunk. It is the responsibility of the client to check for this condition, assemble a broken-up packet from its parts, and not to re-queue any buffer chunks in which as yet unread packet parts reside.

struct **fw_cdev_event_iso_resource**
Iso resources were allocated or freed

Definition

```
struct fw_cdev_event_iso_resource {
  __u64 closure;
  __u32 type;
  __u32 handle;
  __s32 channel;
  __s32 bandwidth;
};
```

Members

closure See fw_cdev_event_common; set by``FW_CDEV_IOC_(DE)ALLOCATE_ISO_RESOURC
ioctl

type FW_CDEV_EVENT_ISO_RESOURCE_ALLOCATED or FW_CDEV_EVENT_ISO_RESOURCE_DEALLOCATE

handle Reference by which an allocated resource can be deallocated

channel Isochronous channel which was (de)allocated, if any

bandwidth Bandwidth allocation units which were (de)allocated, if any

Description

An FW_CDEV_EVENT_ISO_RESOURCE_ALLOCATED event is sent after an isochronous resource was allocated at the IRM. The client has to check **channel** and **bandwidth** for whether the allocation actually succeeded.

An FW_CDEV_EVENT_ISO_RESOURCE_DEALLOCATED event is sent after an isochronous resource was deallocated at the IRM. It is also sent when automatic reallocation after a bus reset failed.

channel is <0 if no channel was (de)allocated or if reallocation failed. **bandwidth** is 0 if no bandwidth was (de)allocated or if reallocation failed.

struct **fw_cdev_event_phy_packet**
A PHY packet was transmitted or received

Definition

```
struct fw_cdev_event_phy_packet {
  __u64 closure;
  __u32 type;
  __u32 rcode;
  __u32 length;
  __u32 data[0];
};
```

Members

closure See fw_cdev_event_common; set by FW_CDEV_IOC_SEND_PHY_PACKET or FW_CDEV_IOC_RECEIVE_PHY_PACKETS ioctl

type FW_CDEV_EVENT_PHY_PACKET_SENT or %⋯_RECEIVED

rcode RCODE_⋯, indicates success or failure of transmission

length Data length in bytes

data Incoming data

Description

If **type** is FW_CDEV_EVENT_PHY_PACKET_SENT, **length** is 0 and **data** empty, except in case of a ping packet: Then, **length** is 4, and **data**[0] is the ping time in 49.152MHz clocks if **rcode** is RCODE_COMPLETE.

If **type** is FW_CDEV_EVENT_PHY_PACKET_RECEIVED, **length** is 8 and **data** consists of the two PHY packet quadlets, in host byte order.

union **fw_cdev_event**
Convenience union of fw_cdev_event_* types

Definition

```
union fw_cdev_event {
  struct fw_cdev_event_common             common;
  struct fw_cdev_event_bus_reset          bus_reset;
  struct fw_cdev_event_response           response;
  struct fw_cdev_event_request            request;
  struct fw_cdev_event_request2           request2;
  struct fw_cdev_event_iso_interrupt      iso_interrupt;
  struct fw_cdev_event_iso_interrupt_mc   iso_interrupt_mc;
  struct fw_cdev_event_iso_resource       iso_resource;
  struct fw_cdev_event_phy_packet         phy_packet;
};
```

Members

common Valid for all types

bus_reset Valid if **common.type** == FW_CDEV_EVENT_BUS_RESET

response Valid if **common.type** == FW_CDEV_EVENT_RESPONSE

request Valid if **common.type** == FW_CDEV_EVENT_REQUEST

request2 Valid if **common.type** == FW_CDEV_EVENT_REQUEST2

iso_interrupt Valid if **common.type** == FW_CDEV_EVENT_ISO_INTERRUPT

iso_interrupt_mc Valid if **common.type** == FW_CDEV_EVENT_ISO_INTERRUPT_MULTICHANNEL

iso_resource Valid if **common.type** == FW_CDEV_EVENT_ISO_RESOURCE_ALLOCATED or FW_CDEV_EVENT_ISO_RESOURCE_DEALLOCATED

phy_packet Valid if **common.type** == FW_CDEV_EVENT_PHY_PACKET_SENT or FW_CDEV_EVENT_PHY_PACKET_RECEIVED

Description

Convenience union for userspace use. Events could be read(2) into an appropriately aligned char buffer and then cast to this union for further processing. Note that for a request, response or iso_interrupt event, the data[] or header[] may make the size of the full event larger than sizeof(union fw_cdev_event). Also note that if you attempt to read(2) an event into a buffer that is not large enough for it, the data that does not fit will be discarded so that the next read(2) will return a new event.

struct **fw_cdev_get_info**
General purpose information ioctl

Definition

```
struct fw_cdev_get_info {
  __u32 version;
  __u32 rom_length;
  __u64 rom;
  __u64 bus_reset;
  __u64 bus_reset_closure;
  __u32 card;
};
```

Members

version The version field is just a running serial number. Both an input parameter (ABI version implemented by the client) and output parameter (ABI version implemented by the kernel). A client shall fill in the ABI **version** for which the client was implemented. This is necessary for forward compatibility.

rom_length If **rom** is non-zero, up to **rom_length** bytes of Configuration ROM will be copied into that user space address. In either case, **rom_length** is updated with the actual length of the Configuration ROM.

rom If non-zero, address of a buffer to be filled by a copy of the device's Configuration ROM

bus_reset If non-zero, address of a buffer to be filled by a `struct fw_cdev_event_bus_reset` with the current state of the bus. This does not cause a bus reset to happen.

bus_reset_closure Value of `closure` in this and subsequent bus reset events

card The index of the card this device belongs to

Description

The `FW_CDEV_IOC_GET_INFO` ioctl is usually the very first one which a client performs right after it opened a /dev/fw* file.

As a side effect, reception of `FW_CDEV_EVENT_BUS_RESET` events to be read(2) is started by this ioctl.

struct **fw_cdev_send_request**
Send an asynchronous request packet

Definition

```
struct fw_cdev_send_request {
  __u32 tcode;
  __u32 length;
  __u64 offset;
  __u64 closure;
  __u64 data;
  __u32 generation;
};
```

Members

tcode Transaction code of the request

length Length of outgoing payload, in bytes

offset 48-bit offset at destination node

closure Passed back to userspace in the response event

data Userspace pointer to payload

generation The bus generation where packet is valid

Description

Send a request to the device. This ioctl implements all outgoing requests. Both quadlet and block request specify the payload as a pointer to the data in the **data** field. Once the transaction completes, the kernel writes an `fw_cdev_event_response` event back. The **closure** field is passed back to user space in the response event.

struct **fw_cdev_send_response**
Send an asynchronous response packet

Definition

```
struct fw_cdev_send_response {
  __u32 rcode;
  __u32 length;
  __u64 data;
  __u32 handle;
};
```

Members

rcode Response code as determined by the userspace handler

length Length of outgoing payload, in bytes

data Userspace pointer to payload

handle The handle from the `fw_cdev_event_request`

Description

Send a response to an incoming request. By setting up an address range using the `FW_CDEV_IOC_ALLOCATE` ioctl, userspace can listen for incoming requests. An incoming request will generate an `FW_CDEV_EVENT_REQUEST`, and userspace must send a reply using this ioctl. The event has a handle to the kernel-side pending transaction, which should be used with this ioctl.

struct **fw_cdev_allocate**
Allocate a CSR in an address range

Definition

```
struct fw_cdev_allocate {
  __u64 offset;
  __u64 closure;
  __u32 length;
  __u32 handle;
  __u64 region_end;
};
```

Members

offset Start offset of the address range

closure To be passed back to userspace in request events

length Length of the CSR, in bytes

handle Handle to the allocation, written by the kernel

region_end First address above the address range (added in ABI v4, 2.6.36)

Description

Allocate an address range in the 48-bit address space on the local node (the controller). This allows userspace to listen for requests with an offset within that address range. Every time when the kernel receives a request within the range, an `fw_cdev_event_request2` event will be emitted. (If the kernel or the client implements ABI version <= 3, an `fw_cdev_event_request` will be generated instead.)

The **closure** field is passed back to userspace in these request events. The **handle** field is an out parameter, returning a handle to the allocated range to be used for later deallocation of the range.

The address range is allocated on all local nodes. The address allocation is exclusive except for the FCP command and response registers. If an exclusive address region is already in use, the ioctl fails with errno set to `EBUSY`.

If kernel and client implement ABI version >= 4, the kernel looks up a free spot of size **length** inside [**offset**..**region_end**) and, if found, writes the start address of the new CSR back in **offset**. I.e. **offset** is an in and out parameter. If this automatic placement of a CSR in a bigger address range is not desired, the client simply needs to set **region_end** = **offset** + **length**.

If the kernel or the client implements ABI version <= 3, **region_end** is ignored and effectively assumed to be **offset** + **length**.

region_end is only present in a kernel header >= 2.6.36. If necessary, this can for example be tested by #ifdef FW_CDEV_EVENT_REQUEST2.

struct **fw_cdev_deallocate**
Free a CSR address range or isochronous resource

Definition

```
struct fw_cdev_deallocate {
  __u32 handle;
};
```

Members

handle Handle to the address range or iso resource, as returned by the kernel when the range or resource was allocated

struct **fw_cdev_initiate_bus_reset**
Initiate a bus reset

Definition

```
struct fw_cdev_initiate_bus_reset {
  __u32 type;
};
```

Members

type FW_CDEV_SHORT_RESET or FW_CDEV_LONG_RESET

Description

Initiate a bus reset for the bus this device is on. The bus reset can be either the original (long) bus reset or the arbitrated (short) bus reset introduced in 1394a-2000.

The ioctl returns immediately. A subsequent fw_cdev_event_bus_reset indicates when the reset actually happened. Since ABI v4, this may be considerably later than the ioctl because the kernel ensures a grace period between subsequent bus resets as per IEEE 1394 bus management specification.

struct **fw_cdev_add_descriptor**
Add contents to the local node's config ROM

Definition

```
struct fw_cdev_add_descriptor {
  __u32 immediate;
  __u32 key;
  __u64 data;
  __u32 length;
  __u32 handle;
};
```

Members

immediate If non-zero, immediate key to insert before pointer

key Upper 8 bits of root directory pointer

data Userspace pointer to contents of descriptor block

length Length of descriptor block data, in quadlets

handle Handle to the descriptor, written by the kernel

Description

Add a descriptor block and optionally a preceding immediate key to the local node's Configuration ROM.

The **key** field specifies the upper 8 bits of the descriptor root directory pointer and the **data** and **length** fields specify the contents. The **key** should be of the form 0xXX000000. The offset part of the root directory entry will be filled in by the kernel.

If not 0, the **immediate** field specifies an immediate key which will be inserted before the root directory pointer.

immediate, **key**, and **data** array elements are CPU-endian quadlets.

If successful, the kernel adds the descriptor and writes back a **handle** to the kernel-side object to be used for later removal of the descriptor block and immediate key. The kernel will also generate a bus reset to signal the change of the Configuration ROM to other nodes.

This ioctl affects the Configuration ROMs of all local nodes. The ioctl only succeeds on device files which represent a local node.

struct **fw_cdev_remove_descriptor**
Remove contents from the Configuration ROM

Definition

```
struct fw_cdev_remove_descriptor {
  __u32 handle;
};
```

Members

handle Handle to the descriptor, as returned by the kernel when the descriptor was added

Description

Remove a descriptor block and accompanying immediate key from the local nodes' Configuration ROMs. The kernel will also generate a bus reset to signal the change of the Configuration ROM to other nodes.

struct **fw_cdev_create_iso_context**
Create a context for isochronous I/O

Definition

```
struct fw_cdev_create_iso_context {
  __u32 type;
  __u32 header_size;
  __u32 channel;
  __u32 speed;
  __u64 closure;
  __u32 handle;
};
```

Members

type FW_CDEV_ISO_CONTEXT_TRANSMIT or FW_CDEV_ISO_CONTEXT_RECEIVE or FW_CDEV_ISO_CONTEXT_RECEIVE_MULTICHANNEL

header_size Header size to strip in single-channel reception

channel Channel to bind to in single-channel reception or transmission

speed Transmission speed

closure To be returned in `fw_cdev_event_iso_interrupt` or `fw_cdev_event_iso_interrupt_multichannel`

handle Handle to context, written back by kernel

Description

Prior to sending or receiving isochronous I/O, a context must be created. The context records information about the transmit or receive configuration and typically maps to an underlying hardware resource. A context is set up for either sending or receiving. It is bound to a specific isochronous **channel**.

In case of multichannel reception, **header_size** and **channel** are ignored and the channels are selected by `FW_CDEV_IOC_SET_ISO_CHANNELS`.

For `FW_CDEV_ISO_CONTEXT_RECEIVE` contexts, **header_size** must be at least 4 and must be a multiple of 4. It is ignored in other context types.

speed is ignored in receive context types.

If a context was successfully created, the kernel writes back a handle to the context, which must be passed in for subsequent operations on that context.

Limitations: No more than one iso context can be created per fd. The total number of contexts that all userspace and kernelspace drivers can create on a card at a time is a hardware limit, typically 4 or 8 contexts per direction, and of them at most one multichannel receive context.

struct **fw_cdev_set_iso_channels**
Select channels in multichannel reception

Definition

```
struct fw_cdev_set_iso_channels {
  __u64 channels;
  __u32 handle;
};
```

Members

channels Bitmask of channels to listen to

handle Handle of the mutichannel receive context

Description

channels is the bitwise or of 1ULL << n for each channel n to listen to.

The ioctl fails with errno `EBUSY` if there is already another receive context on a channel in **channels**. In that case, the bitmask of all unoccupied channels is returned in **channels**.

struct **fw_cdev_iso_packet**
Isochronous packet

Definition

```
struct fw_cdev_iso_packet {
  __u32 control;
  __u32 header[0];
};
```

Members

control Contains the header length (8 uppermost bits), the sy field (4 bits), the tag field (2 bits), a sync flag or a skip flag (1 bit), an interrupt flag (1 bit), and the payload length (16 lowermost bits)

header Header and payload in case of a transmit context.

Description

`struct fw_cdev_iso_packet` is used to describe isochronous packet queues. Use the FW_CDEV_ISO_* macros to fill in **control**. The **header** array is empty in case of receive contexts.

Context type `FW_CDEV_ISO_CONTEXT_TRANSMIT`:

control.HEADER_LENGTH must be a multiple of 4. It specifies the numbers of bytes in **header** that will be prepended to the packet's payload. These bytes are copied into the kernel and will not be accessed after the ioctl has returned.

The **control.SY** and TAG fields are copied to the iso packet header. These fields are specified by IEEE 1394a and IEC 61883-1.

The **control.SKIP** flag specifies that no packet is to be sent in a frame. When using this, all other fields except **control.INTERRUPT** must be zero.

When a packet with the **control.INTERRUPT** flag set has been completed, an `fw_cdev_event_iso_interrupt` event will be sent.

Context type `FW_CDEV_ISO_CONTEXT_RECEIVE`:

control.HEADER_LENGTH must be a multiple of the context's header_size. If the HEADER_LENGTH is larger than the context's header_size, multiple packets are queued for this entry.

The **control.SY** and TAG fields are ignored.

If the **control.SYNC** flag is set, the context drops all packets until a packet with a sy field is received which matches `fw_cdev_start_iso.sync`.

control.PAYLOAD_LENGTH defines how many payload bytes can be received for one packet (in addition to payload quadlets that have been defined as headers and are stripped and returned in the `fw_cdev_event_iso_interrupt` structure). If more bytes are received, the additional bytes are dropped. If less bytes are received, the remaining bytes in this part of the payload buffer will not be written to, not even by the next packet. I.e., packets received in consecutive frames will not necessarily be consecutive in memory. If an entry has queued multiple packets, the PAYLOAD_LENGTH is divided equally among them.

When a packet with the **control.INTERRUPT** flag set has been completed, an `fw_cdev_event_iso_interrupt` event will be sent. An entry that has queued multiple receive packets is completed when its last packet is completed.

Context type `FW_CDEV_ISO_CONTEXT_RECEIVE_MULTICHANNEL`:

Here, `fw_cdev_iso_packet` would be more aptly named _iso_buffer_chunk since it specifies a chunk of the mmap()'ed buffer, while the number and alignment of packets to be placed into the buffer chunk is not known beforehand.

control.PAYLOAD_LENGTH is the size of the buffer chunk and specifies room for header, payload, padding, and trailer bytes of one or more packets. It must be a multiple of 4.

control.HEADER_LENGTH, TAG and SY are ignored. SYNC is treated as described for single-channel reception.

When a buffer chunk with the **control.INTERRUPT** flag set has been filled entirely, an `fw_cdev_event_iso_interrupt_mc` event will be sent.

struct **fw_cdev_queue_iso**
Queue isochronous packets for I/O

Definition

```
struct fw_cdev_queue_iso {
  __u64 packets;
  __u64 data;
  __u32 size;
  __u32 handle;
};
```

Members

packets Userspace pointer to an array of `fw_cdev_iso_packet`

data Pointer into mmap()'ed payload buffer

size Size of the **packets** array, in bytes

handle Isochronous context handle

Description

Queue a number of isochronous packets for reception or transmission. This ioctl takes a pointer to an array of `fw_cdev_iso_packet` structs, which describe how to transmit from or receive into a contiguous region of a mmap()'ed payload buffer. As part of transmit packet descriptors, a series of headers can be supplied, which will be prepended to the payload during DMA.

The kernel may or may not queue all packets, but will write back updated values of the **packets**, **data** and **size** fields, so the ioctl can be resubmitted easily.

In case of a multichannel receive context, **data** must be quadlet-aligned relative to the buffer start.

struct **fw_cdev_start_iso**
Start an isochronous transmission or reception

Definition

```
struct fw_cdev_start_iso {
  __s32 cycle;
  __u32 sync;
  __u32 tags;
```

(continues on next page)

(continued from previous page)

```
  __u32 handle;
};
```

Members

cycle Cycle in which to start I/O. If **cycle** is greater than or equal to 0, the I/O will start on that cycle.

sync Determines the value to wait for for receive packets that have the FW_CDEV_ISO_SYNC bit set

tags Tag filter bit mask. Only valid for isochronous reception. Determines the tag values for which packets will be accepted. Use FW_CDEV_ISO_CONTEXT_MATCH_* macros to set **tags**.

handle Isochronous context handle within which to transmit or receive

struct **fw_cdev_stop_iso**
Stop an isochronous transmission or reception

Definition

```
struct fw_cdev_stop_iso {
  __u32 handle;
};
```

Members

handle Handle of isochronous context to stop

struct **fw_cdev_flush_iso**
flush completed iso packets

Definition

```
struct fw_cdev_flush_iso {
  __u32 handle;
};
```

Members

handle handle of isochronous context to flush

Description

For FW_CDEV_ISO_CONTEXT_TRANSMIT or FW_CDEV_ISO_CONTEXT_RECEIVE contexts, report any completed packets.

For FW_CDEV_ISO_CONTEXT_RECEIVE_MULTICHANNEL contexts, report the current offset in the receive buffer, if it has changed; this is typically in the middle of some buffer chunk.

Any FW_CDEV_EVENT_ISO_INTERRUPT or FW_CDEV_EVENT_ISO_INTERRUPT_MULTICHANNEL events generated by this ioctl are sent synchronously, i.e., are available for reading from the file descriptor when this ioctl returns.

struct **fw_cdev_get_cycle_timer**
read cycle timer register

Definition

```
struct fw_cdev_get_cycle_timer {
  __u64 local_time;
  __u32 cycle_timer;
};
```

Members

local_time system time, in microseconds since the Epoch

cycle_timer Cycle Time register contents

Description

Same as FW_CDEV_IOC_GET_CYCLE_TIMER2, but fixed to use CLOCK_REALTIME and only with microseconds resolution.

In version 1 and 2 of the ABI, this ioctl returned unreliable (non- monotonic) **cycle_timer** values on certain controllers.

struct **fw_cdev_get_cycle_timer2**
read cycle timer register

Definition

```
struct fw_cdev_get_cycle_timer2 {
  __s64 tv_sec;
  __s32 tv_nsec;
  __s32 clk_id;
  __u32 cycle_timer;
};
```

Members

tv_sec system time, seconds

tv_nsec system time, sub-seconds part in nanoseconds

clk_id input parameter, clock from which to get the system time

cycle_timer Cycle Time register contents

Description

The FW_CDEV_IOC_GET_CYCLE_TIMER2 ioctl reads the isochronous cycle timer and also the system clock. This allows to correlate reception time of isochronous packets with system time.

clk_id lets you choose a clock like with POSIX' clock_gettime function. Supported **clk_id** values are POSIX' CLOCK_REALTIME and CLOCK_MONOTONIC and Linux' CLOCK_MONOTONIC_RAW.

cycle_timer consists of 7 bits cycleSeconds, 13 bits cycleCount, and 12 bits cycleOffset, in host byte order. Cf. the Cycle Time register per IEEE 1394 or Isochronous Cycle Timer register per OHCI-1394.

struct **fw_cdev_allocate_iso_resource**
(De)allocate a channel or bandwidth

Definition

```
struct fw_cdev_allocate_iso_resource {
  __u64 closure;
  __u64 channels;
  __u32 bandwidth;
  __u32 handle;
};
```

Members

closure Passed back to userspace in corresponding iso resource events

channels Isochronous channels of which one is to be (de)allocated

bandwidth Isochronous bandwidth units to be (de)allocated

handle Handle to the allocation, written by the kernel (only valid in case of `FW_CDEV_IOC_ALLOCATE_ISO_RESOURCE` ioctls)

Description

The `FW_CDEV_IOC_ALLOCATE_ISO_RESOURCE` ioctl initiates allocation of an isochronous channel and/or of isochronous bandwidth at the isochronous resource manager (IRM). Only one of the channels specified in **channels** is allocated. An `FW_CDEV_EVENT_ISO_RESOURCE_ALLOCATED` is sent after communication with the IRM, indicating success or failure in the event data. The kernel will automatically reallocate the resources after bus resets. Should a reallocation fail, an `FW_CDEV_EVENT_ISO_RESOURCE_DEALLOCATED` event will be sent. The kernel will also automatically deallocate the resources when the file descriptor is closed.

The `FW_CDEV_IOC_DEALLOCATE_ISO_RESOURCE` ioctl can be used to initiate deallocation of resources which were allocated as described above. An `FW_CDEV_EVENT_ISO_RESOURCE_DEALLOCATED` event concludes this operation.

The `FW_CDEV_IOC_ALLOCATE_ISO_RESOURCE_ONCE` ioctl is a variant of allocation without automatic re- or deallocation. An `FW_CDEV_EVENT_ISO_RESOURCE_ALLOCATED` event concludes this operation, indicating success or failure in its data.

The `FW_CDEV_IOC_DEALLOCATE_ISO_RESOURCE_ONCE` ioctl works like `FW_CDEV_IOC_ALLOCATE_ISO_RESOURCE_ONCE` except that resources are freed instead of allocated. An `FW_CDEV_EVENT_ISO_RESOURCE_DEALLOCATED` event concludes this operation.

To summarize, `FW_CDEV_IOC_ALLOCATE_ISO_RESOURCE` allocates iso resources for the lifetime of the fd or **handle**. In contrast, `FW_CDEV_IOC_ALLOCATE_ISO_RESOURCE_ONCE` allocates iso resources for the duration of a bus generation.

channels is a host-endian bitfield with the least significant bit representing channel 0 and the most significant bit representing channel 63: 1ULL << c for each channel c that is a candidate for (de)allocation.

bandwidth is expressed in bandwidth allocation units, i.e. the time to send one quadlet of data (payload or header data) at speed S1600.

struct **fw_cdev_send_stream_packet**
send an asynchronous stream packet

Definition

```
struct fw_cdev_send_stream_packet {
  __u32 length;
  __u32 tag;
  __u32 channel;
  __u32 sy;
  __u64 closure;
  __u64 data;
  __u32 generation;
  __u32 speed;
};
```

Members

length Length of outgoing payload, in bytes

tag Data format tag

channel Isochronous channel to transmit to

sy Synchronization code

closure Passed back to userspace in the response event

data Userspace pointer to payload

generation The bus generation where packet is valid

speed Speed to transmit at

Description

The FW_CDEV_IOC_SEND_STREAM_PACKET ioctl sends an asynchronous stream packet to every device which is listening to the specified channel. The kernel writes an fw_cdev_event_response event which indicates success or failure of the transmission.

struct **fw_cdev_send_phy_packet**
send a PHY packet

Definition

```
struct fw_cdev_send_phy_packet {
  __u64 closure;
  __u32 data[2];
  __u32 generation;
};
```

Members

closure Passed back to userspace in the PHY-packet-sent event

data First and second quadlet of the PHY packet

generation The bus generation where packet is valid

Description

The FW_CDEV_IOC_SEND_PHY_PACKET ioctl sends a PHY packet to all nodes on the same card as this device. After transmission, an FW_CDEV_EVENT_PHY_PACKET_SENT event is generated.

The payload **data**[] shall be specified in host byte order. Usually, **data**[1] needs to be the bitwise inverse of **data**[0]. VersaPHY packets are an exception to this rule.

The ioctl is only permitted on device files which represent a local node.

struct **fw_cdev_receive_phy_packets**
start reception of PHY packets

Definition

```
struct fw_cdev_receive_phy_packets {
  __u64 closure;
};
```

Members

closure Passed back to userspace in phy packet events

Description

This ioctl activates issuing of FW_CDEV_EVENT_PHY_PACKET_RECEIVED due to incoming PHY packets from any node on the same bus as the device.

The ioctl is only permitted on device files which represent a local node.

21.3 Firewire device probing and sysfs interfaces

```
What:           /sys/bus/firewire/devices/fw[0-9]+/
Date:           May 2007
KernelVersion:  2.6.22
Contact:        linux1394-devel@lists.sourceforge.net
Description:
                IEEE 1394 node device attributes.
                Read-only.  Mutable during the node device's␣
↪lifetime.
                See IEEE 1212 for semantic definitions.

                config_rom
                        Contents of the Configuration ROM register.
                        Binary attribute; an array of host-endian␣
↪u32.

                guid
                        The node's EUI-64 in the bus information␣
↪block of
                        Configuration ROM.
                        Hexadecimal string representation of an u64.


What:           /sys/bus/firewire/devices/fw[0-9]+/units
Date:           June 2009
KernelVersion:  2.6.31
Contact:        linux1394-devel@lists.sourceforge.net
Description:
```

```
                IEEE 1394 node device attribute.
                Read-only.  Mutable during the node device's␣
↪lifetime.
                See IEEE 1212 for semantic definitions.

                units
                        Summary of all units present in an IEEE␣
↪1394 node.
                        Contains space-separated tuples of␣
↪specifier_id and
                        version of each unit present in the node. ␣
↪Specifier_id
                        and version are hexadecimal string␣
↪representations of
                        u24 of the respective unit directory␣
↪entries.
                        Specifier_id and version within each tuple␣
↪are separated
                        by a colon.

Users:          udev rules to set ownership and access permissions␣
↪or ACLs of
                /dev/fw[0-9]+ character device files


What:           /sys/bus/firewire/devices/fw[0-9]+/is_local
Date:           July 2012
KernelVersion:  3.6
Contact:        linux1394-devel@lists.sourceforge.net
Description:
                IEEE 1394 node device attribute.
                Read-only and immutable.
Values:         1: The sysfs entry represents a local node (a␣
↪controller card).
                0: The sysfs entry represents a remote node.


What:           /sys/bus/firewire/devices/fw[0-9]+[.][0-9]+/
Date:           May 2007
KernelVersion:  2.6.22
Contact:        linux1394-devel@lists.sourceforge.net
Description:
                IEEE 1394 unit device attributes.
                Read-only.  Immutable during the unit device's␣
↪lifetime.
                See IEEE 1212 for semantic definitions.

                modalias
                        Same as MODALIAS in the uevent at device␣
↪creation.
```

```
                rom_index
                        Offset of the unit directory within the␣
↪parent device's
                        (node device's) Configuration ROM, in␣
↪quadlets.
                        Decimal string representation.


What:           /sys/bus/firewire/devices/*/
Date:           May 2007
KernelVersion:  2.6.22
Contact:        linux1394-devel@lists.sourceforge.net
Description:
                Attributes common to IEEE 1394 node devices and␣
↪unit devices.
                Read-only.  Mutable during the node device's␣
↪lifetime.
                Immutable during the unit device's lifetime.
                See IEEE 1212 for semantic definitions.

                These attributes are only created if the root␣
↪directory of an
                IEEE 1394 node or the unit directory of an IEEE␣
↪1394 unit
                actually contains according entries.

                hardware_version
                        Hexadecimal string representation of an u24.

                hardware_version_name
                        Contents of a respective textual descriptor␣
↪leaf.

                model
                        Hexadecimal string representation of an u24.

                model_name
                        Contents of a respective textual descriptor␣
↪leaf.

                specifier_id
                        Hexadecimal string representation of an u24.
                        Mandatory in unit directories according to␣
↪IEEE 1212.

                vendor
                        Hexadecimal string representation of an u24.
                        Mandatory in the root directory according␣
↪to IEEE 1212.

                vendor_name
```

```
                        Contents of a respective textual descriptor␣
↪leaf.

                version
                        Hexadecimal string representation of an u24.
                        Mandatory in unit directories according to␣
↪IEEE 1212.


What:           /sys/bus/firewire/drivers/sbp2/fw*/host*/target*/
↪*:*:*:*/ieee1394_id
                formerly
                /sys/bus/ieee1394/drivers/sbp2/fw*/host*/target*/
↪*:*:*:*/ieee1394_id
Date:           Feb 2004
KernelVersion:  2.6.4
Contact:        linux1394-devel@lists.sourceforge.net
Description:
                SCSI target port identifier and logical unit␣
↪identifier of a
                logical unit of an SBP-2 target.  The identifiers␣
↪are specified
                in SAM-2...SAM-4 annex A.  They are persistent and␣
↪world-wide
                unique properties the SBP-2 attached target.

                Read-only attribute, immutable during the target's␣
↪lifetime.
                Format, as exposed by firewire-sbp2 since 2.6.22,␣
↪May 2007:
                Colon-separated hexadecimal string representations␣
↪of
                        u64 EUI-64 : u24 directory_ID : u16 LUN
                without 0x prefixes, without whitespace.  The␣
↪former sbp2 driver
                (removed in 2.6.37 after being superseded by␣
↪firewire-sbp2) used
                a somewhat shorter format which was not as close to␣
↪SAM.

Users:          udev rules to create /dev/disk/by-id/ symlinks
```

int **fw_csr_string**(const u32 * directory, int key, char * buf, size_t size)
reads a string from the configuration ROM

Parameters

const u32 * directory e.g. root directory or unit directory

int key the key of the preceding directory entry

char * buf where to put the string

size_t size size of **buf**, in bytes

Description

The string is taken from a minimal ASCII text descriptor leaf after the immediate entry with **key**. The string is zero-terminated. An overlong string is silently truncated such that it and the zero byte fit into **size**.

Returns strlen(buf) or a negative error code.

21.4 Firewire core transaction interfaces

void **fw_send_request**(struct fw_card * card, struct fw_transaction * t, int tcode, int destination_id, int generation, int speed, unsigned long long offset, void * payload, size_t length, fw_transaction_callback_t callback, void * callback_data)

submit a request packet for transmission

Parameters

struct fw_card * card interface to send the request at

struct fw_transaction * t transaction instance to which the request belongs

int tcode transaction code

int destination_id destination node ID, consisting of bus_ID and phy_ID

int generation bus generation in which request and response are valid

int speed transmission speed

unsigned long long offset 48bit wide offset into destination's address space

void * payload data payload for the request subaction

size_t length length of the payload, in bytes

fw_transaction_callback_t callback function to be called when the transaction is completed

void * callback_data data to be passed to the transaction completion callback

Description

Submit a request packet into the asynchronous request transmission queue. Can be called from atomic context. If you prefer a blocking API, use `fw_run_transaction()` in a context that can sleep.

In case of lock requests, specify one of the firewire-core specific `TCODE_` constants instead of `TCODE_LOCK_REQUEST` in **tcode**.

Make sure that the value in **destination_id** is not older than the one in **generation**. Otherwise the request is in danger to be sent to a wrong node.

In case of asynchronous stream packets i.e. `TCODE_STREAM_DATA`, the caller needs to synthesize **destination_id** with fw_stream_packet_destination_id(). It will contain tag, channel, and sy data instead of a node ID then.

The payload buffer at **data** is going to be DMA-mapped except in case of **length** <= 8 or of local (loopback) requests. Hence make sure that the buffer complies with the restrictions of the streaming DMA mapping API. **payload** must not be freed before the **callback** is called.

In case of request types without payload, **data** is NULL and **length** is 0.

After the transaction is completed successfully or unsuccessfully, the **callback** will be called. Among its parameters is the response code which is either one of the rcodes per IEEE 1394 or, in case of internal errors, the firewire-core specific `RCODE_SEND_ERROR`. The other firewire-core specific rcodes (`RCODE_CANCELLED`, `RCODE_BUSY`, `RCODE_GENERATION`, `RCODE_NO_ACK`) denote transaction timeout, busy responder, stale request generation, or missing ACK respectively.

Note some timing corner cases: `fw_send_request()` may complete much earlier than when the request packet actually hits the wire. On the other hand, transaction completion and hence execution of **callback** may happen even before `fw_send_request()` returns.

int **fw_run_transaction**(struct fw_card * card, int tcode, int destination_id, int generation, int speed, unsigned long long offset, void * payload, size_t length)

send request and sleep until transaction is completed

Parameters

struct fw_card * card card interface for this request

int tcode transaction code

int destination_id destination node ID, consisting of bus_ID and phy_ID

int generation bus generation in which request and response are valid

int speed transmission speed

unsigned long long offset 48bit wide offset into destination's address space

void * payload data payload for the request subaction

size_t length length of the payload, in bytes

Description

Returns the RCODE. See `fw_send_request()` for parameter documentation. Unlike `fw_send_request()`, **data** points to the payload of the request or/and to the payload of the response. DMA mapping restrictions apply to outbound request payloads of >= 8 bytes but not to inbound response payloads.

int **fw_core_add_address_handler**(struct fw_address_handler * handler, const struct fw_address_region * region)

register for incoming requests

Parameters

struct fw_address_handler * handler callback

const struct fw_address_region * region region in the IEEE 1212 node space address range

Description

region->start, ->end, and handler->length have to be quadlet-aligned.

When a request is received that falls within the specified address range, the specified callback is invoked. The parameters passed to the callback give the details of the particular request.

To be called in process context. Return value: 0 on success, non-zero otherwise.

The start offset of the handler's address region is determined by `fw_core_add_address_handler()` and is returned in handler->offset.

Address allocations are exclusive, except for the FCP registers.

void **fw_core_remove_address_handler**(struct fw_address_handler * handler)
unregister an address handler

Parameters

struct fw_address_handler * handler callback

Description

To be called in process context.

When `fw_core_remove_address_handler()` returns, **handler->callback()** is guaranteed to not run on any CPU anymore.

int **fw_get_request_speed**(struct fw_request * request)
returns speed at which the **request** was received

Parameters

struct fw_request * request firewire request data

const char * **fw_rcode_string**(int rcode)
convert a firewire result code to an error description

Parameters

int rcode the result code

21.5 Firewire Isochronous I/O interfaces

void **fw_iso_resource_manage**(struct fw_card * card, int generation, u64 channels_mask, int * channel, int * bandwidth, bool allocate)
Allocate or deallocate a channel and/or bandwidth

Parameters

struct fw_card * card card interface for this action

int generation bus generation

u64 channels_mask bitmask for channel allocation

int * channel pointer for returning channel allocation result

int * bandwidth pointer for returning bandwidth allocation result

bool allocate whether to allocate (true) or deallocate (false)

Description

In parameters: card, generation, channels_mask, bandwidth, allocate Out parameters: channel, bandwidth

This function blocks (sleeps) during communication with the IRM.

Allocates or deallocates at most one channel out of channels_mask. channels_mask is a bitfield with MSB for channel 63 and LSB for channel 0. (Note, the IRM's CHANNELS_AVAILABLE is a big-endian bitfield with MSB for channel 0 and LSB for channel 63.) Allocates or deallocates as many bandwidth allocation units as specified.

Returns channel < 0 if no channel was allocated or deallocated. Returns bandwidth = 0 if no bandwidth was allocated or deallocated.

If generation is stale, deallocations succeed but allocations fail with channel = -EAGAIN.

If channel allocation fails, no bandwidth will be allocated either. If bandwidth allocation fails, no channel will be allocated either. But deallocations of channel and bandwidth are tried independently of each other's success.

CHAPTER

TWENTYTWO

THE LINUX PCI DRIVER IMPLEMENTER'S API GUIDE

Table of contents

22.1 PCI Support Library

unsigned char **pci_bus_max_busnr**(struct pci_bus * bus)
returns maximum PCI bus number of given bus' children

Parameters

struct pci_bus * bus pointer to PCI bus structure to search

Description

Given a PCI bus, returns the highest PCI bus number present in the set including the given PCI bus and its list of child PCI buses.

int **pci_status_get_and_clear_errors**(struct pci_dev * pdev)
return and clear error bits in PCI_STATUS

Parameters

struct pci_dev * pdev the PCI device

Description

Returns error bits set in PCI_STATUS and clears them.

int **pci_find_capability**(struct pci_dev * dev, int cap)
query for devices' capabilities

Parameters

struct pci_dev * dev PCI device to query

int cap capability code

Description

Tell if a device supports a given PCI capability. Returns the address of the requested capability structure within the device's PCI configuration space or 0 in case the device does not support it. Possible values for **cap** include:

`PCI_CAP_ID_PM` Power Management `PCI_CAP_ID_AGP` Accelerated Graphics Port `PCI_CAP_ID_VPD` Vital Product Data `PCI_CAP_ID_SLOTID` Slot Identification `PCI_CAP_ID_MSI` Message Signalled Interrupts

PCI_CAP_ID_CHSWP CompactPCI HotSwap PCI_CAP_ID_PCIX PCI-X PCI_CAP_ID_EXP PCI Express

int **pci_bus_find_capability**(struct pci_bus * bus, unsigned int devfn, int cap)
query for devices' capabilities

Parameters

struct pci_bus * bus the PCI bus to query

unsigned int devfn PCI device to query

int cap capability code

Description

Like pci_find_capability() but works for PCI devices that do not have a pci_dev structure set up yet.

Returns the address of the requested capability structure within the device's PCI configuration space or 0 in case the device does not support it.

int **pci_find_next_ext_capability**(struct pci_dev * dev, int start, int cap)
Find an extended capability

Parameters

struct pci_dev * dev PCI device to query

int start address at which to start looking (0 to start at beginning of list)

int cap capability code

Description

Returns the address of the next matching extended capability structure within the device's PCI configuration space or 0 if the device does not support it. Some capabilities can occur several times, e.g., the vendor-specific capability, and this provides a way to find them all.

int **pci_find_ext_capability**(struct pci_dev * dev, int cap)
Find an extended capability

Parameters

struct pci_dev * dev PCI device to query

int cap capability code

Description

Returns the address of the requested extended capability structure within the device's PCI configuration space or 0 if the device does not support it. Possible values for **cap** include:

PCI_EXT_CAP_ID_ERR Advanced Error Reporting PCI_EXT_CAP_ID_VC Virtual Channel PCI_EXT_CAP_ID_DSN Device Serial Number PCI_EXT_CAP_ID_PWR Power Budgeting

u64 **pci_get_dsn**(struct pci_dev * dev)
Read and return the 8-byte Device Serial Number

Parameters

struct pci_dev * dev PCI device to query

Description

Looks up the PCI_EXT_CAP_ID_DSN and reads the 8 bytes of the Device Serial Number.

Returns the DSN, or zero if the capability does not exist.

int **pci_find_next_ht_capability**(struct pci_dev * dev, int pos, int ht_cap)
query a device's Hypertransport capabilities

Parameters

struct pci_dev * dev PCI device to query

int pos Position from which to continue searching

int ht_cap Hypertransport capability code

Description

To be used in conjunction with `pci_find_ht_capability()` to search for all capabilities matching **ht_cap**. **pos** should always be a value returned from `pci_find_ht_capability()`.

NB. To be 100% safe against broken PCI devices, the caller should take steps to avoid an infinite loop.

int **pci_find_ht_capability**(struct pci_dev * dev, int ht_cap)
query a device's Hypertransport capabilities

Parameters

struct pci_dev * dev PCI device to query

int ht_cap Hypertransport capability code

Description

Tell if a device supports a given Hypertransport capability. Returns an address within the device's PCI configuration space or 0 in case the device does not support the request capability. The address points to the PCI capability, of type PCI_CAP_ID_HT, which has a Hypertransport capability matching **ht_cap**.

struct resource * **pci_find_parent_resource**(const struct pci_dev * dev,
struct resource * res)
return resource region of parent bus of given region

Parameters

const struct pci_dev * dev PCI device structure contains resources to be searched

struct resource * res child resource record for which parent is sought

Description

For given resource region of given device, return the resource region of parent bus the given region is contained in.

struct resource * **pci_find_resource**(struct pci_dev * dev, struct resource * res)
Return matching PCI device resource

Parameters

struct pci_dev * dev PCI device to query

struct resource * res Resource to look for

Description

Goes over standard PCI resources (BARs) and checks if the given resource is partially or fully contained in any of them. In that case the matching resource is returned, NULL otherwise.

int **pci_platform_power_transition**(struct pci_dev * dev, pci_power_t state)
Use platform to change device power state

Parameters

struct pci_dev * dev PCI device to handle.

pci_power_t state State to put the device into.

int **pci_set_power_state**(struct pci_dev * dev, pci_power_t state)
Set the power state of a PCI device

Parameters

struct pci_dev * dev PCI device to handle.

pci_power_t state PCI power state (D0, D1, D2, D3hot) to put the device into.

Description

Transition a device to a new power state, using the platform firmware and/or the device's PCI PM registers.

RETURN VALUE: -EINVAL if the requested state is invalid. -EIO if device does not support PCI PM or its PM capabilities register has a wrong version, or device doesn't support the requested state. 0 if the transition is to D1 or D2 but D1 and D2 are not supported. 0 if device already is in the requested state. 0 if the transition is to D3 but D3 is not supported. 0 if device's power state has been successfully changed.

pci_power_t **pci_choose_state**(struct pci_dev * dev, pm_message_t state)
Choose the power state of a PCI device

Parameters

struct pci_dev * dev PCI device to be suspended

pm_message_t state target sleep state for the whole system. This is the value that is passed to suspend() function.

Description

Returns PCI power state suitable for given device and given system message.

int **pci_save_state**(struct pci_dev * dev)
save the PCI configuration space of a device before suspending

Parameters

struct pci_dev * dev PCI device that we're dealing with

void **pci_restore_state**(struct pci_dev * dev)
Restore the saved state of a PCI device

Parameters

struct pci_dev * dev PCI device that we're dealing with

struct pci_saved_state * **pci_store_saved_state**(struct pci_dev * dev)
Allocate and return an opaque struct containing the device saved state.

Parameters

struct pci_dev * dev PCI device that we're dealing with

Description

Return NULL if no state or error.

int **pci_load_saved_state**(struct pci_dev * dev, struct pci_saved_state * state)
Reload the provided save state into struct pci_dev.

Parameters

struct pci_dev * dev PCI device that we're dealing with

struct pci_saved_state * state Saved state returned from `pci_store_saved_state()`

int **pci_load_and_free_saved_state**(struct pci_dev * dev, struct pci_saved_state ** state)
Reload the save state pointed to by state, and free the memory allocated for it.

Parameters

struct pci_dev * dev PCI device that we're dealing with

struct pci_saved_state ** state Pointer to saved state returned from `pci_store_saved_state()`

int **pci_reenable_device**(struct pci_dev * dev)
Resume abandoned device

Parameters

struct pci_dev * dev PCI device to be resumed

NOTE

This function is a backend of pci_default_resume() and is not supposed to be called by normal code, write proper resume handler and use it instead.

int **pci_enable_device_io**(struct pci_dev * dev)
Initialize a device for use with IO space

Parameters

struct pci_dev * dev PCI device to be initialized

Description

Initialize device before it's used by a driver. Ask low-level code to enable I/O resources. Wake up the device if it was suspended. Beware, this function can fail.

int **pci_enable_device_mem**(struct pci_dev * dev)
Initialize a device for use with Memory space

Parameters

struct pci_dev * dev PCI device to be initialized

Description

Initialize device before it's used by a driver. Ask low-level code to enable Memory resources. Wake up the device if it was suspended. Beware, this function can fail.

int **pci_enable_device**(struct pci_dev * dev)
Initialize device before it's used by a driver.

Parameters

struct pci_dev * dev PCI device to be initialized

Description

Initialize device before it's used by a driver. Ask low-level code to enable I/O and memory. Wake up the device if it was suspended. Beware, this function can fail.

Note we don't actually enable the device many times if we call this function repeatedly (we just increment the count).

int **pcim_enable_device**(struct pci_dev * pdev)
Managed `pci_enable_device()`

Parameters

struct pci_dev * pdev PCI device to be initialized

Description

Managed `pci_enable_device()`.

void **pcim_pin_device**(struct pci_dev * pdev)
Pin managed PCI device

Parameters

struct pci_dev * pdev PCI device to pin

Description

Pin managed PCI device **pdev**. Pinned device won't be disabled on driver detach. **pdev** must have been enabled with `pcim_enable_device()`.

void **pci_disable_device**(struct pci_dev * dev)
Disable PCI device after use

Parameters

struct pci_dev * dev PCI device to be disabled

Description

Signal to the system that the PCI device is not in use by the system anymore. This only involves disabling PCI bus-mastering, if active.

Note we don't actually disable the device until all callers of `pci_enable_device()` have called `pci_disable_device()`.

int **pci_set_pcie_reset_state**(struct pci_dev * dev, enum pcie_reset_state state)
set reset state for device dev

Parameters

struct pci_dev * dev the PCIe device reset

enum pcie_reset_state state Reset state to enter into

Description

Sets the PCI reset state for the device.

bool **pci_pme_capable**(struct pci_dev * dev, pci_power_t state)
check the capability of PCI device to generate PME#

Parameters

struct pci_dev * dev PCI device to handle.

pci_power_t state PCI state from which device will issue PME#.

void **pci_pme_active**(struct pci_dev * dev, bool enable)
enable or disable PCI device's PME# function

Parameters

struct pci_dev * dev PCI device to handle.

bool enable 'true' to enable PME# generation; 'false' to disable it.

Description

The caller must verify that the device is capable of generating PME# before calling this function with **enable** equal to 'true'.

int **pci_enable_wake**(struct pci_dev * pci_dev, pci_power_t state, bool enable)
change wakeup settings for a PCI device

Parameters

struct pci_dev * pci_dev Target device

pci_power_t state PCI state from which device will issue wakeup events

bool enable Whether or not to enable event generation

Description

If **enable** is set, check device_may_wakeup() for the device before calling __pci_enable_wake() for it.

int **pci_wake_from_d3**(struct pci_dev * dev, bool enable)
enable/disable device to wake up from D3_hot or D3_cold

Parameters

struct pci_dev * dev PCI device to prepare

bool enable True to enable wake-up event generation; false to disable

Description

Many drivers want the device to wake up the system from D3_hot or D3_cold and this function allows them to set that up cleanly - `pci_enable_wake()` should not be called twice in a row to enable wake-up due to PCI PM vs ACPI ordering constraints.

This function only returns error code if the device is not allowed to wake up the system from sleep or it is not capable of generating PME# from both D3_hot and D3_cold and the platform is unable to enable wake-up power for it.

int **pci_prepare_to_sleep**(struct pci_dev * dev)
prepare PCI device for system-wide transition into a sleep state

Parameters

struct pci_dev * dev Device to handle.

Description

Choose the power state appropriate for the device depending on whether it can wake up the system and/or is power manageable by the platform (PCI_D3hot is the default) and put the device into that state.

int **pci_back_from_sleep**(struct pci_dev * dev)
turn PCI device on during system-wide transition into working state

Parameters

struct pci_dev * dev Device to handle.

Description

Disable device's system wake-up capability and put it into D0.

bool **pci_dev_run_wake**(struct pci_dev * dev)
Check if device can generate run-time wake-up events.

Parameters

struct pci_dev * dev Device to check.

Description

Return true if the device itself is capable of generating wake-up events (through the platform or using the native PCIe PME) or if the device supports PME and one of its upstream bridges can generate wake-up events.

void **pci_d3cold_enable**(struct pci_dev * dev)
Enable D3cold for device

Parameters

struct pci_dev * dev PCI device to handle

Description

This function can be used in drivers to enable D3cold from the device they handle. It also updates upstream PCI bridge PM capabilities accordingly.

void **pci_d3cold_disable**(struct pci_dev * dev)
Disable D3cold for device

Parameters

struct pci_dev * dev PCI device to handle

Description

This function can be used in drivers to disable D3cold from the device they handle. It also updates upstream PCI bridge PM capabilities accordingly.

int **pci_enable_atomic_ops_to_root**(struct pci_dev * dev, u32 cap_mask)
enable AtomicOp requests to root port

Parameters

struct pci_dev * dev the PCI device

u32 cap_mask mask of desired AtomicOp sizes, including one or more of: PCI_EXP_DEVCAP2_ATOMIC_COMP32 PCI_EXP_DEVCAP2_ATOMIC_COMP64 PCI_EXP_DEVCAP2_ATOMIC_COMP128

Description

Return 0 if all upstream bridges support AtomicOp routing, egress blocking is disabled on all upstream ports, and the root port supports the requested completion capabilities (32-bit, 64-bit and/or 128-bit AtomicOp completion), or negative otherwise.

u8 **pci_common_swizzle**(struct pci_dev * dev, u8 * pinp)
swizzle INTx all the way to root bridge

Parameters

struct pci_dev * dev the PCI device

u8 * pinp pointer to the INTx pin value (1=INTA, 2=INTB, 3=INTD, 4=INTD)

Description

Perform INTx swizzling for a device. This traverses through all PCI-to-PCI bridges all the way up to a PCI root bus.

void **pci_release_region**(struct pci_dev * pdev, int bar)
Release a PCI bar

Parameters

struct pci_dev * pdev PCI device whose resources were previously reserved by `pci_request_region()`

int bar BAR to release

Description

Releases the PCI I/O and memory resources previously reserved by a successful call to `pci_request_region()`. Call this function only after all use of the PCI regions has ceased.

int **pci_request_region**(struct pci_dev * pdev, int bar, const char * res_name)
Reserve PCI I/O and memory resource

Parameters

struct pci_dev * pdev PCI device whose resources are to be reserved

int bar BAR to be reserved

const char * res_name Name to be associated with resource

Description

Mark the PCI region associated with PCI device **pdev** BAR **bar** as being reserved by owner **res_name**. Do not access any address inside the PCI regions unless this call returns successfully.

Returns 0 on success, or EBUSY on error. A warning message is also printed on failure.

void **pci_release_selected_regions**(struct pci_dev * pdev, int bars)
Release selected PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources were previously reserved

int bars Bitmask of BARs to be released

Description

Release selected PCI I/O and memory resources previously reserved. Call this function only after all use of the PCI regions has ceased.

int **pci_request_selected_regions**(struct pci_dev * pdev, int bars, const char * res_name)
Reserve selected PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources are to be reserved

int bars Bitmask of BARs to be requested

const char * res_name Name to be associated with resource

void **pci_release_regions**(struct pci_dev * pdev)
Release reserved PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources were previously reserved by pci_request_regions()

Description

Releases all PCI I/O and memory resources previously reserved by a successful call to pci_request_regions(). Call this function only after all use of the PCI regions has ceased.

int **pci_request_regions**(struct pci_dev * pdev, const char * res_name)
Reserve PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources are to be reserved

const char * res_name Name to be associated with resource.

Description

Mark all PCI regions associated with PCI device **pdev** as being reserved by owner **res_name**. Do not access any address inside the PCI regions unless this call returns successfully.

Returns 0 on success, or EBUSY on error. A warning message is also printed on failure.

int **pci_request_regions_exclusive**(struct pci_dev * pdev, const char * res_name)
Reserve PCI I/O and memory resources

Parameters

struct pci_dev * pdev PCI device whose resources are to be reserved

const char * res_name Name to be associated with resource.

Description

Mark all PCI regions associated with PCI device **pdev** as being reserved by owner **res_name**. Do not access any address inside the PCI regions unless this call returns successfully.

pci_request_regions_exclusive() will mark the region so that /dev/mem and the sysfs MMIO access will not be allowed.

Returns 0 on success, or EBUSY on error. A warning message is also printed on failure.

int **pci_remap_iospace**(const struct resource * res, phys_addr_t phys_addr)
Remap the memory mapped I/O space

Parameters

const struct resource * res Resource describing the I/O space

phys_addr_t phys_addr physical address of range to be mapped

Description

Remap the memory mapped I/O space described by the **res** and the CPU physical address **phys_addr** into virtual address space. Only architectures that have memory mapped IO functions defined (and the PCI_IOBASE value defined) should call this function.

void **pci_unmap_iospace**(struct resource * res)
Unmap the memory mapped I/O space

Parameters

struct resource * res resource to be unmapped

Description

Unmap the CPU virtual address **res** from virtual address space. Only architectures that have memory mapped IO functions defined (and the PCI_IOBASE value defined) should call this function.

int **devm_pci_remap_iospace**(struct device * dev, const struct resource * res, phys_addr_t phys_addr)
Managed `pci_remap_iospace()`

Parameters

struct device * dev Generic device to remap IO address for

const struct resource * res Resource describing the I/O space

phys_addr_t phys_addr physical address of range to be mapped

Description

Managed `pci_remap_iospace()`. Map is automatically unmapped on driver detach.

void __iomem * **devm_pci_remap_cfgspace**(struct device * dev, resource_size_t offset, resource_size_t size)
Managed pci_remap_cfgspace()

Parameters

struct device * dev Generic device to remap IO address for

resource_size_t offset Resource address to map

resource_size_t size Size of map

Description

Managed pci_remap_cfgspace(). Map is automatically unmapped on driver detach.

void __iomem * **devm_pci_remap_cfg_resource**(struct device * dev, struct resource * res)
check, request region and ioremap cfg resource

Parameters

struct device * dev generic device to handle the resource for

struct resource * res configuration space resource to be handled

Description

Checks that a resource is a valid memory region, requests the memory region and ioremaps with pci_remap_cfgspace() API that ensures the proper PCI configuration space memory attributes are guaranteed.

All operations are managed and will be undone on driver detach.

Returns a pointer to the remapped memory or an ERR_PTR() encoded error code on failure. Usage example:

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = devm_pci_remap_cfg_resource(&pdev->dev, res);
if (IS_ERR(base))
        return PTR_ERR(base);
```

void **pci_set_master**(struct pci_dev * dev)
enables bus-mastering for device dev

Parameters

struct pci_dev * dev the PCI device to enable

Description

Enables bus-mastering on the device and calls pcibios_set_master() to do the needed arch specific settings.

void **pci_clear_master**(struct pci_dev * dev)
disables bus-mastering for device dev

Parameters

struct pci_dev * dev the PCI device to disable

int **pci_set_cacheline_size**(struct pci_dev * dev)
ensure the CACHE_LINE_SIZE register is programmed

Parameters

struct pci_dev * dev the PCI device for which MWI is to be enabled

Description

Helper function for pci_set_mwi. Originally copied from drivers/net/acenic.c. Copyright 1998-2001 by Jes Sorensen, <jes**trained**-monkey.org>.

Return

An appropriate -ERRNO error value on error, or zero for success.

int **pci_set_mwi**(struct pci_dev * dev)
enables memory-write-invalidate PCI transaction

Parameters

struct pci_dev * dev the PCI device for which MWI is enabled

Description

Enables the Memory-Write-Invalidate transaction in `PCI_COMMAND`.

Return

An appropriate -ERRNO error value on error, or zero for success.

int **pcim_set_mwi**(struct pci_dev * dev)
a device-managed `pci_set_mwi()`

Parameters

struct pci_dev * dev the PCI device for which MWI is enabled

Description

Managed `pci_set_mwi()`.

Return

An appropriate -ERRNO error value on error, or zero for success.

int **pci_try_set_mwi**(struct pci_dev * dev)
enables memory-write-invalidate PCI transaction

Parameters

struct pci_dev * dev the PCI device for which MWI is enabled

Description

Enables the Memory-Write-Invalidate transaction in `PCI_COMMAND`. Callers are not required to check the return value.

Return

An appropriate -ERRNO error value on error, or zero for success.

void **pci_clear_mwi**(struct pci_dev * dev)
disables Memory-Write-Invalidate for device dev

Parameters

struct pci_dev * dev the PCI device to disable

Description

Disables PCI Memory-Write-Invalidate transaction on the device

void **pci_intx**(struct pci_dev * pdev, int enable)
enables/disables PCI INTx for device dev

Parameters

struct pci_dev * pdev the PCI device to operate on

int enable boolean: whether to enable or disable PCI INTx

Description

Enables/disables PCI INTx for device **pdev**

bool **pci_check_and_mask_intx**(struct pci_dev * dev)
mask INTx on pending interrupt

Parameters

struct pci_dev * dev the PCI device to operate on

Description

Check if the device dev has its INTx line asserted, mask it and return true in that case. False is returned if no interrupt was pending.

bool **pci_check_and_unmask_intx**(struct pci_dev * dev)
unmask INTx if no interrupt is pending

Parameters

struct pci_dev * dev the PCI device to operate on

Description

Check if the device dev has its INTx line asserted, unmask it if not and return true. False is returned and the mask remains active if there was still an interrupt pending.

int **pci_wait_for_pending_transaction**(struct pci_dev * dev)
wait for pending transaction

Parameters

struct pci_dev * dev the PCI device to operate on

Description

Return 0 if transaction is pending 1 otherwise.

bool **pcie_has_flr**(struct pci_dev * dev)
check if a device supports function level resets

Parameters

struct pci_dev * dev device to check

Description

Returns true if the device advertises support for PCIe function level resets.

int **pcie_flr**(struct pci_dev * dev)
initiate a PCIe function level reset

Parameters

struct pci_dev * dev device to reset

Description

Initiate a function level reset on **dev**. The caller should ensure the device supports FLR before calling this function, e.g. by using the `pcie_has_flr()` helper.

int **pci_bridge_secondary_bus_reset**(struct pci_dev * dev)
Reset the secondary bus on a PCI bridge.

Parameters

struct pci_dev * dev Bridge device

Description

Use the bridge control register to assert reset on the secondary bus. Devices on the secondary bus are left in power-on state.

int **__pci_reset_function_locked**(struct pci_dev * dev)
reset a PCI device function while holding the **dev** mutex lock.

Parameters

struct pci_dev * dev PCI device to reset

Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

The device function is presumed to be unused and the caller is holding the device mutex lock when this function is called.

Resetting the device will make the contents of PCI configuration space random, so any caller of this must be prepared to reinitialise the device including MSI, bus mastering, BARs, decoding IO and memory spaces, etc.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

int **pci_reset_function**(struct pci_dev * dev)
quiesce and reset a PCI device function

Parameters

struct pci_dev * dev PCI device to reset

Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

This function does not just reset the PCI portion of a device, but clears all the state associated with the device. This function differs from `__pci_reset_function_locked()` in that it saves and restores device state over the reset and takes the PCI device lock.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

int **pci_reset_function_locked**(struct pci_dev * dev)
quiesce and reset a PCI device function

Parameters

struct pci_dev * dev PCI device to reset

Description

Some devices allow an individual function to be reset without affecting other functions in the same device. The PCI device must be responsive to PCI config space in order to use this function.

This function does not just reset the PCI portion of a device, but clears all the state associated with the device. This function differs from `__pci_reset_function_locked()` in that it saves and restores device state over the reset. It also differs from `pci_reset_function()` in that it requires the PCI device lock to be held.

Returns 0 if the device function was successfully reset or negative if the device doesn't support resetting a single function.

int **pci_try_reset_function**(struct pci_dev * dev)
quiesce and reset a PCI device function

Parameters

struct pci_dev * dev PCI device to reset

Description

Same as above, except return -EAGAIN if unable to lock device.

int **pci_probe_reset_slot**(struct pci_slot * slot)
probe whether a PCI slot can be reset

Parameters

struct pci_slot * slot PCI slot to probe

Description

Return 0 if slot can be reset, negative if a slot reset is not supported.

int **pci_probe_reset_bus**(struct pci_bus * bus)
probe whether a PCI bus can be reset

Parameters

struct pci_bus * bus PCI bus to probe

Description

Return 0 if bus can be reset, negative if a bus reset is not supported.

int **pci_reset_bus**(struct pci_dev * pdev)
Try to reset a PCI bus

Parameters

struct pci_dev * pdev top level PCI device to reset via slot/bus

Description

Same as above except return -EAGAIN if the bus cannot be locked

int **pcix_get_max_mmrbc**(struct pci_dev * dev)
get PCI-X maximum designed memory read byte count

Parameters

struct pci_dev * dev PCI device to query

Description

Returns mmrbc: maximum designed memory read count in bytes or appropriate error value.

int **pcix_get_mmrbc**(struct pci_dev * dev)
get PCI-X maximum memory read byte count

Parameters

struct pci_dev * dev PCI device to query

Description

Returns mmrbc: maximum memory read count in bytes or appropriate error value.

int **pcix_set_mmrbc**(struct pci_dev * dev, int mmrbc)
set PCI-X maximum memory read byte count

Parameters

struct pci_dev * dev PCI device to query

int mmrbc maximum memory read count in bytes valid values are 512, 1024, 2048, 4096

Description

If possible sets maximum memory read byte count, some bridges have errata that prevent this.

int **pcie_get_readrq**(struct pci_dev * dev)
get PCI Express read request size

Parameters

struct pci_dev * dev PCI device to query

Description

Returns maximum memory read request in bytes or appropriate error value.

int **pcie_set_readrq**(struct pci_dev * dev, int rq)
set PCI Express maximum memory read request

Parameters

struct pci_dev * dev PCI device to query

int rq maximum memory read count in bytes valid values are 128, 256, 512, 1024, 2048, 4096

Description

If possible sets maximum memory read request in bytes

int **pcie_get_mps**(struct pci_dev * dev)
get PCI Express maximum payload size

Parameters

struct pci_dev * dev PCI device to query

Description

Returns maximum payload size in bytes

int **pcie_set_mps**(struct pci_dev * dev, int mps)
set PCI Express maximum payload size

Parameters

struct pci_dev * dev PCI device to query

int mps maximum payload size in bytes valid values are 128, 256, 512, 1024, 2048, 4096

Description

If possible sets maximum payload size

u32 **pcie_bandwidth_available**(struct pci_dev * dev, struct pci_dev ** limiting_dev, enum pci_bus_speed * speed, enum pcie_link_width * width)
determine minimum link settings of a PCIe device and its bandwidth limitation

Parameters

struct pci_dev * dev PCI device to query

struct pci_dev ** limiting_dev storage for device causing the bandwidth limitation

enum pci_bus_speed * speed storage for speed of limiting device

enum pcie_link_width * width storage for width of limiting device

Description

Walk up the PCI device chain and find the point where the minimum bandwidth is available. Return the bandwidth available there and (if limiting_dev, speed, and width pointers are supplied) information about that point. The bandwidth returned is in Mb/s, i.e., megabits/second of raw bandwidth.

enum pci_bus_speed **pcie_get_speed_cap**(struct pci_dev * dev)
query for the PCI device's link speed capability

Parameters

struct pci_dev * dev PCI device to query

Description

Query the PCI device speed capability. Return the maximum link speed supported by the device.

enum pcie_link_width **pcie_get_width_cap**(struct pci_dev * dev)
query for the PCI device's link width capability

Parameters

struct pci_dev * dev PCI device to query

Description

Query the PCI device width capability. Return the maximum link width supported by the device.

void **pcie_print_link_status**(struct pci_dev * dev)
Report the PCI device's link speed and width

Parameters

struct pci_dev * dev PCI device to query

Description

Report the available bandwidth at the device.

int **pci_select_bars**(struct pci_dev * dev, unsigned long flags)
Make BAR mask from the type of resource

Parameters

struct pci_dev * dev the PCI device for which BAR mask is made

unsigned long flags resource type mask to be selected

Description

This helper routine makes bar mask from the type of resource.

int **pci_add_dynid**(struct pci_driver * drv, unsigned int vendor, unsigned int device, unsigned int subvendor, unsigned int subdevice, unsigned int class, unsigned int class_mask, unsigned long driver_data)
add a new PCI device ID to this driver and re-probe devices

Parameters

struct pci_driver * drv target pci driver

unsigned int vendor PCI vendor ID

unsigned int device PCI device ID

unsigned int subvendor PCI subvendor ID

unsigned int subdevice PCI subdevice ID

unsigned int class PCI class

unsigned int class_mask PCI class mask

unsigned long driver_data private driver data

Description

Adds a new dynamic pci device ID to this driver and causes the driver to probe for all devices again. **drv** must have been registered prior to calling this function.

Context

Does GFP_KERNEL allocation.

Return

0 on success, -errno on failure.

const struct pci_device_id * **pci_match_id**(const struct pci_device_id * ids, struct pci_dev * dev)
See if a pci device matches a given pci_id table

Parameters

const struct pci_device_id * ids array of PCI device id structures to search in

struct pci_dev * dev the PCI device structure to match against.

Description

Used by a driver to check whether a PCI device present in the system is in its list of supported devices. Returns the matching pci_device_id structure or `NULL` if there is no match.

Deprecated, don't use this as it will not catch any dynamic ids that a driver might want to check for.

int **__pci_register_driver**(struct pci_driver * drv, struct module * owner, const char * mod_name)
register a new pci driver

Parameters

struct pci_driver * drv the driver structure to register

struct module * owner owner module of drv

const char * mod_name module name string

Description

Adds the driver structure to the list of registered drivers. Returns a negative value on error, otherwise 0. If no error occurred, the driver remains registered even if no device was claimed during registration.

void **pci_unregister_driver**(struct pci_driver * drv)
unregister a pci driver

Parameters

struct pci_driver * drv the driver structure to unregister

Description

Deletes the driver structure from the list of registered PCI drivers, gives it a chance to clean up by calling its remove() function for each device it was responsible for, and marks those devices as driverless.

struct pci_driver * **pci_dev_driver**(const struct pci_dev * dev)
get the pci_driver of a device

Parameters

const struct pci_dev * dev the device to query

Description

Returns the appropriate pci_driver structure or NULL if there is no registered driver for the device.

struct pci_dev * **pci_dev_get**(struct pci_dev * dev)
increments the reference count of the pci device structure

Parameters

struct pci_dev * dev the device being referenced

Description

Each live reference to a device should be refcounted.

Drivers for PCI devices should normally record such references in their probe() methods, when they bind to a device, and release them by calling pci_dev_put(), in their disconnect() methods.

A pointer to the device with the incremented reference counter is returned.

void **pci_dev_put**(struct pci_dev * dev)
release a use of the pci device structure

Parameters

struct pci_dev * dev device that's been disconnected

Description

Must be called when a user of a device is finished with it. When the last user of the device calls this function, the memory of the device is freed.

void **pci_stop_and_remove_bus_device**(struct pci_dev * dev)
remove a PCI device and any children

Parameters

struct pci_dev * dev the device to remove

Description

Remove a PCI device from the device lists, informing the drivers that the device has been removed. We also remove any subordinate buses and children in a depth-first manner.

For each device we remove, delete the device structure from the device lists, remove the /proc entry, and notify userspace (/sbin/hotplug).

struct pci_bus * **pci_find_bus**(int domain, int busnr)
locate PCI bus from a given domain and bus number

Parameters

int domain number of PCI domain to search

int busnr number of desired PCI bus

Description

Given a PCI bus number and domain number, the desired PCI bus is located in the global list of PCI buses. If the bus is found, a pointer to its data structure is returned. If no bus is found, `NULL` is returned.

struct pci_bus * **pci_find_next_bus**(const struct pci_bus * from)
begin or continue searching for a PCI bus

Parameters

const struct pci_bus * from Previous PCI bus found, or `NULL` for new search.

Description

Iterates through the list of known PCI buses. A new search is initiated by passing `NULL` as the **from** argument. Otherwise if **from** is not `NULL`, searches continue from next device on the global list.

struct pci_dev * **pci_get_slot**(struct pci_bus * bus, unsigned int devfn)
locate PCI device for a given PCI slot

Parameters

struct pci_bus * bus PCI bus on which desired PCI device resides

unsigned int devfn encodes number of PCI slot in which the desired PCI device resides and the logical device number within that slot in case of multi-function devices.

Description

Given a PCI bus and slot/function number, the desired PCI device is located in the list of PCI devices. If the device is found, its reference count is increased and this function returns a pointer to its data structure. The caller must decrement the reference count by calling `pci_dev_put()`. If no device is found, `NULL` is returned.

struct pci_dev * **pci_get_domain_bus_and_slot**(int domain, unsigned int bus, unsigned int devfn)
locate PCI device for a given PCI domain (segment), bus, and slot

Parameters

int domain PCI domain/segment on which the PCI device resides.

unsigned int bus PCI bus on which desired PCI device resides

unsigned int devfn encodes number of PCI slot in which the desired PCI device resides and the logical device number within that slot in case of multi-function devices.

Description

Given a PCI domain, bus, and slot/function number, the desired PCI device is located in the list of PCI devices. If the device is found, its reference count is increased and this function returns a pointer to its data structure. The caller must decrement the reference count by calling `pci_dev_put()`. If no device is found, `NULL` is returned.

struct pci_dev * **pci_get_subsys**(unsigned int vendor, unsigned int device, unsigned int ss_vendor, unsigned int ss_device, struct pci_dev * from)
begin or continue searching for a PCI device by vendor/subvendor/device/subdevice id

Parameters

unsigned int vendor PCI vendor id to match, or `PCI_ANY_ID` to match all vendor ids

unsigned int device PCI device id to match, or `PCI_ANY_ID` to match all device ids

unsigned int ss_vendor PCI subsystem vendor id to match, or `PCI_ANY_ID` to match all vendor ids

unsigned int ss_device PCI subsystem device id to match, or `PCI_ANY_ID` to match all device ids

struct pci_dev * from Previous PCI device found in search, or `NULL` for new search.

Description

Iterates through the list of known PCI devices. If a PCI device is found with a matching **vendor**, **device**, **ss_vendor** and **ss_device**, a pointer to its device structure is returned, and the reference count to the device is incremented. Otherwise, `NULL` is returned. A new search is initiated by passing `NULL` as the **from** argument. Otherwise if **from** is not `NULL`, searches continue from next device on the global list. The reference count for **from** is always decremented if it is not `NULL`.

struct pci_dev * **pci_get_device**(unsigned int vendor, unsigned int device, struct pci_dev * from)
begin or continue searching for a PCI device by vendor/device id

Parameters

unsigned int vendor PCI vendor id to match, or PCI_ANY_ID to match all vendor ids

unsigned int device PCI device id to match, or PCI_ANY_ID to match all device ids

struct pci_dev * from Previous PCI device found in search, or NULL for new search.

Description

Iterates through the list of known PCI devices. If a PCI device is found with a matching **vendor** and **device**, the reference count to the device is incremented and a pointer to its device structure is returned. Otherwise, NULL is returned. A new search is initiated by passing NULL as the **from** argument. Otherwise if **from** is not NULL, searches continue from next device on the global list. The reference count for **from** is always decremented if it is not NULL.

struct pci_dev * **pci_get_class**(unsigned int class, struct pci_dev * from)
begin or continue searching for a PCI device by class

Parameters

unsigned int class search for a PCI device with this class designation

struct pci_dev * from Previous PCI device found in search, or NULL for new search.

Description

Iterates through the list of known PCI devices. If a PCI device is found with a matching **class**, the reference count to the device is incremented and a pointer to its device structure is returned. Otherwise, NULL is returned. A new search is initiated by passing NULL as the **from** argument. Otherwise if **from** is not NULL, searches continue from next device on the global list. The reference count for **from** is always decremented if it is not NULL.

int **pci_dev_present**(const struct pci_device_id * ids)
Returns 1 if device matching the device list is present, 0 if not.

Parameters

const struct pci_device_id * ids A pointer to a null terminated list of struct pci_device_id structures that describe the type of PCI device the caller is trying to find.

Description

Obvious fact: You do not have a reference to any device that might be found by this function, so if that device is removed from the system right after this function is finished, the value will be stale. Use this function to find devices that are usually built into a system, or for a general hint as to if another device happens to be present at this specific moment in time.

void **pci_msi_mask_irq**(struct irq_data * data)
Generic IRQ chip callback to mask PCI/MSI interrupts

Parameters

struct irq_data * data pointer to irqdata associated to that interrupt

void **pci_msi_unmask_irq**(struct irq_data * data)
Generic IRQ chip callback to unmask PCI/MSI interrupts

Parameters

struct irq_data * data pointer to irqdata associated to that interrupt

int **pci_msi_vec_count**(struct pci_dev * dev)
Return the number of MSI vectors a device can send

Parameters

struct pci_dev * dev device to report about

Description

This function returns the number of MSI vectors a device requested via Multiple Message Capable register. It returns a negative errno if the device is not capable sending MSI interrupts. Otherwise, the call succeeds and returns a power of two, up to a maximum of 2^5 (32), according to the MSI specification.

int **pci_msix_vec_count**(struct pci_dev * dev)
return the number of device's MSI-X table entries

Parameters

struct pci_dev * dev pointer to the pci_dev data structure of MSI-X device function This function returns the number of device's MSI-X table entries and therefore the number of MSI-X vectors device is capable of sending. It returns a negative errno if the device is not capable of sending MSI-X interrupts.

int **pci_msi_enabled**(void)
is MSI enabled?

Parameters

void no arguments

Description

Returns true if MSI has not been disabled by the command-line option pci=nomsi.

int **pci_enable_msix_range**(struct pci_dev * dev, struct msix_entry * entries, int minvec, int maxvec)
configure device's MSI-X capability structure

Parameters

struct pci_dev * dev pointer to the pci_dev data structure of MSI-X device function

struct msix_entry * entries pointer to an array of MSI-X entries

int minvec minimum number of MSI-X IRQs requested

int maxvec maximum number of MSI-X IRQs requested

Description

Setup the MSI-X capability structure of device function with a maximum possible number of interrupts in the range between **minvec** and **maxvec** upon its software

driver call to request for MSI-X mode enabled on its hardware device function. It returns a negative errno if an error occurs. If it succeeds, it returns the actual number of interrupts allocated and indicates the successful configuration of MSI-X capability structure with new allocated MSI-X interrupts.

int **pci_alloc_irq_vectors_affinity**(struct pci_dev * dev, unsigned int min_vecs, unsigned int max_vecs, unsigned int flags, struct irq_affinity * affd)
allocate multiple IRQs for a device

Parameters

struct pci_dev * dev PCI device to operate on

unsigned int min_vecs minimum number of vectors required (must be >= 1)

unsigned int max_vecs maximum (desired) number of vectors

unsigned int flags flags or quirks for the allocation

struct irq_affinity * affd optional description of the affinity requirements

Description

Allocate up to **max_vecs** interrupt vectors for **dev**, using MSI-X or MSI vectors if available, and fall back to a single legacy vector if neither is available. Return the number of vectors allocated, (which might be smaller than **max_vecs**) if successful, or a negative error code on error. If less than **min_vecs** interrupt vectors are available for **dev** the function will fail with -ENOSPC.

To get the Linux IRQ number used for a vector that can be passed to request_irq() use the `pci_irq_vector()` helper.

void **pci_free_irq_vectors**(struct pci_dev * dev)
free previously allocated IRQs for a device

Parameters

struct pci_dev * dev PCI device to operate on

Description

Undoes the allocations and enabling in pci_alloc_irq_vectors().

int **pci_irq_vector**(struct pci_dev * dev, unsigned int nr)
return Linux IRQ number of a device vector

Parameters

struct pci_dev * dev PCI device to operate on

unsigned int nr device-relative interrupt vector index (0-based).

const struct cpumask * **pci_irq_get_affinity**(struct pci_dev * dev, int nr)
return the affinity of a particular MSI vector

Parameters

struct pci_dev * dev PCI device to operate on

int nr device-relative interrupt vector index (0-based).

struct irq_domain * **pci_msi_create_irq_domain**(struct fwnode_handle * fwnode, struct msi_domain_info * info, struct irq_domain * parent)
Create a MSI interrupt domain

Parameters

struct fwnode_handle * fwnode Optional fwnode of the interrupt controller

struct msi_domain_info * info MSI domain info

struct irq_domain * parent Parent irq domain

Description

Updates the domain and chip ops and creates a MSI interrupt domain.

Return

A domain pointer or NULL in case of failure.

int **pci_bus_alloc_resource**(struct pci_bus * bus, struct resource * res, resource_size_t size, resource_size_t align, resource_size_t min, unsigned long type_mask, resource_size_t (*alignf)(void *, const struct resource *, resource_size_t, resource_size_t), void * alignf_data)
allocate a resource from a parent bus

Parameters

struct pci_bus * bus PCI bus

struct resource * res resource to allocate

resource_size_t size size of resource to allocate

resource_size_t align alignment of resource to allocate

resource_size_t min minimum /proc/iomem address to allocate

unsigned long type_mask IORESOURCE_* type flags

resource_size_t (*)(void *, const struct resource *, resource_size_t, resource_s
resource alignment function

void * alignf_data data argument for resource alignment function

Description

Given the PCI bus a device resides on, the size, minimum address, alignment and type, try to find an acceptable resource allocation for a specific device resource.

void **pci_bus_add_device**(struct pci_dev * dev)
start driver for a single device

Parameters

struct pci_dev * dev device to add

Description

This adds add sysfs entries and start device drivers

void **pci_bus_add_devices**(const struct pci_bus * bus)
start driver for PCI devices

Parameters

const struct pci_bus * bus bus to check for new devices

Description

Start driver for PCI devices and add some sysfs entries.

struct pci_ops * **pci_bus_set_ops**(struct pci_bus * bus, struct pci_ops * ops)
Set raw operations of pci bus

Parameters

struct pci_bus * bus pci bus struct

struct pci_ops * ops new raw operations

Description

Return previous raw operations

void **pci_cfg_access_lock**(struct pci_dev * dev)
Lock PCI config reads/writes

Parameters

struct pci_dev * dev pci device struct

Description

When access is locked, any userspace reads or writes to config space and concurrent lock requests will sleep until access is allowed via `pci_cfg_access_unlock()` again.

bool **pci_cfg_access_trylock**(struct pci_dev * dev)
try to lock PCI config reads/writes

Parameters

struct pci_dev * dev pci device struct

Description

Same as pci_cfg_access_lock, but will return 0 if access is already locked, 1 otherwise. This function can be used from atomic contexts.

void **pci_cfg_access_unlock**(struct pci_dev * dev)
Unlock PCI config reads/writes

Parameters

struct pci_dev * dev pci device struct

Description

This function allows PCI config accesses to resume.

enum pci_lost_interrupt_reason **pci_lost_interrupt**(struct pci_dev * pdev)
reports a lost PCI interrupt

Parameters

struct pci_dev * pdev device whose interrupt is lost

Description

The primary function of this routine is to report a lost interrupt in a standard way which users can recognise (instead of blaming the driver).

Return

a suggestion for fixing it (although the driver is not required to act on this).

int **pci_request_irq**(struct pci_dev * dev, unsigned int nr, irq_handler_t handler, irq_handler_t thread_fn, void * dev_id, const char * fmt, ...)
allocate an interrupt line for a PCI device

Parameters

struct pci_dev * dev PCI device to operate on

unsigned int nr device-relative interrupt vector index (0-based).

irq_handler_t handler Function to be called when the IRQ occurs. Primary handler for threaded interrupts. If NULL and thread_fn != NULL the default primary handler is installed.

irq_handler_t thread_fn Function called from the IRQ handler thread If NULL, no IRQ thread is created

void * dev_id Cookie passed back to the handler function

const char * fmt Printf-like format string naming the handler

... variable arguments

Description

This call allocates interrupt resources and enables the interrupt line and IRQ handling. From the point this call is made **handler** and **thread_fn** may be invoked. All interrupts requested using this function might be shared.

dev_id must not be NULL and must be globally unique.

void **pci_free_irq**(struct pci_dev * dev, unsigned int nr, void * dev_id)
free an interrupt allocated with pci_request_irq

Parameters

struct pci_dev * dev PCI device to operate on

unsigned int nr device-relative interrupt vector index (0-based).

void * dev_id Device identity to free

Description

Remove an interrupt handler. The handler is removed and if the interrupt line is no longer in use by any driver it is disabled. The caller must ensure the interrupt

is disabled on the device before calling this function. The function does not return until any executing interrupts for this IRQ have completed.

This function must not be called from interrupt context.

bool **pcie_relaxed_ordering_enabled**(struct pci_dev * dev)
Probe for PCIe relaxed ordering enable

Parameters

struct pci_dev * dev PCI device to query

Description

Returns true if the device has enabled relaxed ordering attribute.

int **pci_scan_slot**(struct pci_bus * bus, int devfn)
Scan a PCI slot on a bus for devices

Parameters

struct pci_bus * bus PCI bus to scan

int devfn slot number to scan (must have zero function)

Description

Scan a PCI slot on the specified PCI bus for devices, adding discovered devices to the **bus->devices** list. New devices will not have is_added set.

Returns the number of new devices found.

unsigned int **pci_scan_child_bus**(struct pci_bus * bus)
Scan devices below a bus

Parameters

struct pci_bus * bus Bus to scan for devices

Description

Scans devices below **bus** including subordinate buses. Returns new subordinate number including all the found devices.

unsigned int **pci_rescan_bus**(struct pci_bus * bus)
Scan a PCI bus for devices

Parameters

struct pci_bus * bus PCI bus to scan

Description

Scan a PCI bus and child buses for new devices, add them, and enable them.

Returns the max number of subordinate bus discovered.

struct pci_slot * **pci_create_slot**(struct pci_bus * parent, int slot_nr, const char * name, struct hotplug_slot * hotplug)
create or increment refcount for physical PCI slot

Parameters

struct pci_bus * parent struct pci_bus of parent bridge

int slot_nr PCI_SLOT(pci_dev->devfn) or -1 for placeholder

const char * name user visible string presented in /sys/bus/pci/slots/<name>

struct hotplug_slot * hotplug set if caller is hotplug driver, NULL otherwise

Description

PCI slots have first class attributes such as address, speed, width, and a `struct pci_slot` is used to manage them. This interface will either return a new `struct pci_slot` to the caller, or if the pci_slot already exists, its refcount will be incremented.

Slots are uniquely identified by a **pci_bus**, **slot_nr** tuple.

There are known platforms with broken firmware that assign the same name to multiple slots. Workaround these broken platforms by renaming the slots on behalf of the caller. If firmware assigns name N to multiple slots:

The first slot is assigned N The second slot is assigned N-1 The third slot is assigned N-2 etc.

Placeholder slots: In most cases, **pci_bus**, **slot_nr** will be sufficient to uniquely identify a slot. There is one notable exception - pSeries (rpaphp), where the **slot_nr** cannot be determined until a device is actually inserted into the slot. In this scenario, the caller may pass -1 for **slot_nr**.

The following semantics are imposed when the caller passes **slot_nr** == -1. First, we no longer check for an existing `struct` pci_slot, as there may be many slots with **slot_nr** of -1. The other change in semantics is user-visible, which is the 'address' parameter presented in sysfs will consist solely of a dddd:bb tuple, where dddd is the PCI domain of the `struct` pci_bus and bb is the bus number. In other words, the devfn of the 'placeholder' slot will not be displayed.

void **pci_destroy_slot**(struct pci_slot * slot)
decrement refcount for physical PCI slot

Parameters

struct pci_slot * slot struct pci_slot to decrement

Description

`struct` pci_slot is refcounted, so destroying them is really easy; we just call kobject_put on its kobj and let our release methods do the rest.

void **pci_hp_create_module_link**(struct pci_slot * pci_slot)
create symbolic link to the hotplug driver module.

Parameters

struct pci_slot * pci_slot struct pci_slot

Description

Helper function for pci_hotplug_core.c to create symbolic link to the hotplug driver module.

void **pci_hp_remove_module_link**(struct pci_slot * pci_slot)
remove symbolic link to the hotplug driver module.

Parameters

struct pci_slot * pci_slot struct pci_slot

Description

Helper function for pci_hotplug_core.c to remove symbolic link to the hotplug driver module.

int **pci_enable_rom**(struct pci_dev * pdev)
enable ROM decoding for a PCI device

Parameters

struct pci_dev * pdev PCI device to enable

Description

Enable ROM decoding on **dev**. This involves simply turning on the last bit of the PCI ROM BAR. Note that some cards may share address decoders between the ROM and other resources, so enabling it may disable access to MMIO registers or other card memory.

void **pci_disable_rom**(struct pci_dev * pdev)
disable ROM decoding for a PCI device

Parameters

struct pci_dev * pdev PCI device to disable

Description

Disable ROM decoding on a PCI device by turning off the last bit in the ROM BAR.

void __iomem * **pci_map_rom**(struct pci_dev * pdev, size_t * size)
map a PCI ROM to kernel space

Parameters

struct pci_dev * pdev pointer to pci device struct

size_t * size pointer to receive size of pci window over ROM

Return

kernel virtual pointer to image of ROM

Description

Map a PCI ROM into kernel space. If ROM is boot video ROM, the shadow BIOS copy will be returned instead of the actual ROM.

void **pci_unmap_rom**(struct pci_dev * pdev, void __iomem * rom)
unmap the ROM from kernel space

Parameters

struct pci_dev * pdev pointer to pci device struct

void __iomem * rom virtual address of the previous mapping

Description

Remove a mapping of a previously mapped ROM

int **pci_enable_sriov**(struct pci_dev * dev, int nr_virtfn)
enable the SR-IOV capability

Parameters

struct pci_dev * dev the PCI device

int nr_virtfn number of virtual functions to enable

Description

Returns 0 on success, or negative on failure.

void **pci_disable_sriov**(struct pci_dev * dev)
disable the SR-IOV capability

Parameters

struct pci_dev * dev the PCI device

int **pci_num_vf**(struct pci_dev * dev)
return number of VFs associated with a PF device_release_driver

Parameters

struct pci_dev * dev the PCI device

Description

Returns number of VFs, or 0 if SR-IOV is not enabled.

int **pci_vfs_assigned**(struct pci_dev * dev)
returns number of VFs are assigned to a guest

Parameters

struct pci_dev * dev the PCI device

Description

Returns number of VFs belonging to this device that are assigned to a guest. If device is not a physical function returns 0.

int **pci_sriov_set_totalvfs**(struct pci_dev * dev, u16 numvfs)

- reduce the TotalVFs available

Parameters

struct pci_dev * dev the PCI PF device

u16 numvfs number that should be used for TotalVFs supported

Description

Should be called from PF driver's probe routine with device's mutex held.

Returns 0 if PF is an SRIOV-capable device and value of numvfs valid. If not a PF return -ENOSYS; if numvfs is invalid return -EINVAL; if VFs already enabled, return -EBUSY.

int **pci_sriov_get_totalvfs**(struct pci_dev * dev)

- get total VFs supported on this device

Parameters

struct pci_dev * dev the PCI PF device

Description

For a PCIe device with SRIOV support, return the PCIe SRIOV capability value of TotalVFs or the value of driver_max_VFs if the driver reduced it. Otherwise 0.

int **pci_sriov_configure_simple**(struct pci_dev * dev, int nr_virtfn)
helper to configure SR-IOV

Parameters

struct pci_dev * dev the PCI device

int nr_virtfn number of virtual functions to enable, 0 to disable

Description

Enable or disable SR-IOV for devices that don't require any PF setup before enabling SR-IOV. Return value is negative on error, or number of VFs allocated on success.

ssize_t **pci_read_legacy_io**(struct file * filp, struct kobject * kobj, struct bin_attribute * bin_attr, char * buf, loff_t off, size_t count)
read byte(s) from legacy I/O port space

Parameters

struct file * filp open sysfs file

struct kobject * kobj kobject corresponding to file to read from

struct bin_attribute * bin_attr struct bin_attribute for this file

char * buf buffer to store results

loff_t off offset into legacy I/O port space

size_t count number of bytes to read

Description

Reads 1, 2, or 4 bytes from legacy I/O port space using an arch specific callback routine (pci_legacy_read).

ssize_t **pci_write_legacy_io**(struct file * filp, struct kobject * kobj, struct bin_attribute * bin_attr, char * buf, loff_t off, size_t count)
write byte(s) to legacy I/O port space

Parameters

struct file * filp open sysfs file

struct kobject * kobj kobject corresponding to file to read from

struct bin_attribute * bin_attr struct bin_attribute for this file

char * buf buffer containing value to be written

loff_t off offset into legacy I/O port space

size_t count number of bytes to write

Description

Writes 1, 2, or 4 bytes from legacy I/O port space using an arch specific callback routine (pci_legacy_write).

int **pci_mmap_legacy_mem**(struct file * filp, struct kobject * kobj, struct bin_attribute * attr, struct vm_area_struct * vma)
map legacy PCI memory into user memory space

Parameters

struct file * filp open sysfs file

struct kobject * kobj kobject corresponding to device to be mapped

struct bin_attribute * attr struct bin_attribute for this file

struct vm_area_struct * vma struct vm_area_struct passed to mmap

Description

Uses an arch specific callback, pci_mmap_legacy_mem_page_range, to mmap legacy memory space (first meg of bus space) into application virtual memory space.

int **pci_mmap_legacy_io**(struct file * filp, struct kobject * kobj, struct bin_attribute * attr, struct vm_area_struct * vma)
map legacy PCI IO into user memory space

Parameters

struct file * filp open sysfs file

struct kobject * kobj kobject corresponding to device to be mapped

struct bin_attribute * attr struct bin_attribute for this file

struct vm_area_struct * vma struct vm_area_struct passed to mmap

Description

Uses an arch specific callback, pci_mmap_legacy_io_page_range, to mmap legacy IO space (first meg of bus space) into application virtual memory space. Returns -ENOSYS if the operation isn't supported

void **pci_adjust_legacy_attr**(struct pci_bus * b, enum pci_mmap_state mmap_type)
adjustment of legacy file attributes

Parameters

struct pci_bus * b bus to create files under

enum pci_mmap_state mmap_type I/O port or memory

Description

Stub implementation. Can be overridden by arch if necessary.

void **pci_create_legacy_files**(struct pci_bus * b)
create legacy I/O port and memory files

Parameters

struct pci_bus * b bus to create files under

Description

Some platforms allow access to legacy I/O port and ISA memory space on a per-bus basis. This routine creates the files and ties them into their associated read, write and mmap files from pci-sysfs.c

On error unwind, but don't propagate the error to the caller as it is ok to set up the PCI bus without these files.

int **pci_mmap_resource**(struct kobject * kobj, struct bin_attribute * attr, struct vm_area_struct * vma, int write_combine)
map a PCI resource into user memory space

Parameters

struct kobject * kobj kobject for mapping

struct bin_attribute * attr struct bin_attribute for the file being mapped

struct vm_area_struct * vma struct vm_area_struct passed into the mmap

int write_combine 1 for write_combine mapping

Description

Use the regular PCI mapping routines to map a PCI resource into userspace.

void **pci_remove_resource_files**(struct pci_dev * pdev)
cleanup resource files

Parameters

struct pci_dev * pdev dev to cleanup

Description

If we created resource files for **pdev**, remove them from sysfs and free their resources.

int **pci_create_resource_files**(struct pci_dev * pdev)
create resource files in sysfs for **dev**

Parameters

struct pci_dev * pdev dev in question

Description

Walk the resources in **pdev** creating files for each resource available.

ssize_t **pci_write_rom**(struct file * filp, struct kobject * kobj, struct bin_attribute * bin_attr, char * buf, loff_t off, size_t count)
used to enable access to the PCI ROM display

Parameters

struct file * filp sysfs file

struct kobject * kobj kernel object handle

struct bin_attribute * bin_attr struct bin_attribute for this file

char * buf user input

loff_t off file offset

size_t count number of byte in input

Description

writing anything except 0 enables it

ssize_t **pci_read_rom**(struct file * filp, struct kobject * kobj, struct bin_attribute * bin_attr, char * buf, loff_t off, size_t count)
read a PCI ROM

Parameters

struct file * filp sysfs file

struct kobject * kobj kernel object handle

struct bin_attribute * bin_attr struct bin_attribute for this file

char * buf where to put the data we read from the ROM

loff_t off file offset

size_t count number of bytes to read

Description

Put **count** bytes starting at **off** into **buf** from the ROM in the PCI device corresponding to **kobj**.

void **pci_remove_sysfs_dev_files**(struct pci_dev * pdev)
cleanup PCI specific sysfs files

Parameters

struct pci_dev * pdev device whose entries we should free

Description

Cleanup when **pdev** is removed from sysfs.

22.2 PCI Hotplug Support Library

int **__pci_hp_register**(struct hotplug_slot * slot, struct pci_bus * bus, int devnr, const char * name, struct module * owner, const char * mod_name)
register a hotplug_slot with the PCI hotplug subsystem

Parameters

struct hotplug_slot * slot pointer to the `struct hotplug_slot` to register

struct pci_bus * bus bus this slot is on

int devnr device number

const char * name name registered with kobject core

struct module * owner caller module owner

const char * mod_name caller module name

Description

Prepares a hotplug slot for in-kernel use and immediately publishes it to user space in one go. Drivers may alternatively carry out the two steps separately by invoking pci_hp_initialize() and `pci_hp_add()`.

Returns 0 if successful, anything else for an error.

int **__pci_hp_initialize**(struct hotplug_slot * slot, struct pci_bus * bus, int devnr, const char * name, struct module * owner, const char * mod_name)
prepare hotplug slot for in-kernel use

Parameters

struct hotplug_slot * slot pointer to the `struct hotplug_slot` to initialize

struct pci_bus * bus bus this slot is on

int devnr slot number

const char * name name registered with kobject core

struct module * owner caller module owner

const char * mod_name caller module name

Description

Allocate and fill in a PCI slot for use by a hotplug driver. Once this has been called, the driver may invoke hotplug_slot_name() to get the slot's unique name. The driver must be prepared to handle a ->reset_slot callback from this point on.

Returns 0 on success or a negative int on error.

int **pci_hp_add**(struct hotplug_slot * slot)
publish hotplug slot to user space

Parameters

struct hotplug_slot * slot pointer to the `struct hotplug_slot` to publish

Description

Make a hotplug slot's sysfs interface available and inform user space of its addition by sending a uevent. The hotplug driver must be prepared to handle all `struct hotplug_slot_ops` callbacks from this point on.

Returns 0 on success or a negative int on error.

void **pci_hp_deregister**(struct hotplug_slot * slot)
deregister a hotplug_slot with the PCI hotplug subsystem

Parameters

struct hotplug_slot * slot pointer to the `struct hotplug_slot` to deregister

Description

The **slot** must have been registered with the pci hotplug subsystem previously with a call to pci_hp_register().

Returns 0 if successful, anything else for an error.

void **pci_hp_del**(struct hotplug_slot * slot)
unpublish hotplug slot from user space

Parameters

struct hotplug_slot * slot pointer to the `struct hotplug_slot` to unpublish

Description

Remove a hotplug slot's sysfs interface.

Returns 0 on success or a negative int on error.

void **pci_hp_destroy**(struct hotplug_slot * slot)
remove hotplug slot from in-kernel use

Parameters

struct hotplug_slot * slot pointer to the `struct hotplug_slot` to destroy

Description

Destroy a PCI slot used by a hotplug driver. Once this has been called, the driver may no longer invoke hotplug_slot_name() to get the slot's unique name. The driver no longer needs to handle a ->reset_slot callback from this point on.

Returns 0 on success or a negative int on error.

22.3 PCI Peer-to-Peer DMA Support

The PCI bus has pretty decent support for performing DMA transfers between two devices on the bus. This type of transaction is henceforth called Peer-to-Peer (or P2P). However, there are a number of issues that make P2P transactions tricky to do in a perfectly safe way.

One of the biggest issues is that PCI doesn't require forwarding transactions between hierarchy domains, and in PCIe, each Root Port defines a separate hierarchy domain. To make things worse, there is no simple way to determine if a given Root Complex supports this or not. (See PCIe r4.0, sec 1.3.1). Therefore, as of this writing, the kernel only supports doing P2P when the endpoints involved are all behind the same PCI bridge, as such devices are all in the same PCI hierarchy domain, and the spec guarantees that all transactions within the hierarchy will be routable, but it does not require routing between hierarchies.

The second issue is that to make use of existing interfaces in Linux, memory that is used for P2P transactions needs to be backed by struct pages. However, PCI BARs are not typically cache coherent so there are a few corner case gotchas with these pages so developers need to be careful about what they do with them.

22.3.1 Driver Writer's Guide

In a given P2P implementation there may be three or more different types of kernel drivers in play:

- Provider - A driver which provides or publishes P2P resources like memory or doorbell registers to other drivers.
- Client - A driver which makes use of a resource by setting up a DMA transaction to or from it.
- Orchestrator - A driver which orchestrates the flow of data between clients and providers.

In many cases there could be overlap between these three types (i.e., it may be typical for a driver to be both a provider and a client).

For example, in the NVMe Target Copy Offload implementation:

- The NVMe PCI driver is both a client, provider and orchestrator in that it exposes any CMB (Controller Memory Buffer) as a P2P memory resource (provider), it accepts P2P memory pages as buffers in requests to be used directly (client) and it can also make use of the CMB as submission queue entries (orchestrator).
- The RDMA driver is a client in this arrangement so that an RNIC can DMA directly to the memory exposed by the NVMe device.
- The NVMe Target driver (nvmet) can orchestrate the data from the RNIC to the P2P memory (CMB) and then to the NVMe device (and vice versa).

This is currently the only arrangement supported by the kernel but one could imagine slight tweaks to this that would allow for the same functionality. For example, if a specific RNIC added a BAR with some memory behind it, its driver could add support as a P2P provider and then the NVMe Target could use the RNIC's memory instead of the CMB in cases where the NVMe cards in use do not have CMB support.

Provider Drivers

A provider simply needs to register a BAR (or a portion of a BAR) as a P2P DMA resource using `pci_p2pdma_add_resource()`. This will register struct pages for all the specified memory.

After that it may optionally publish all of its resources as P2P memory using `pci_p2pmem_publish()`. This will allow any orchestrator drivers to find and use the memory. When marked in this way, the resource must be regular memory with no side effects.

For the time being this is fairly rudimentary in that all resources are typically going to be P2P memory. Future work will likely expand this to include other types of resources like doorbells.

Client Drivers

A client driver typically only has to conditionally change its DMA map routine to use the mapping function `pci_p2pdma_map_sg()` instead of the usual `dma_map_sg()` function. Memory mapped in this way does not need to be unmapped.

The client may also, optionally, make use of `is_pci_p2pdma_page()` to determine when to use the P2P mapping functions and when to use the regular mapping functions. In some situations, it may be more appropriate to use a flag to indicate a given request is P2P memory and map appropriately. It is important to ensure that struct pages that back P2P memory stay out of code that does not have support for them as other code may treat the pages as regular memory which may not be appropriate.

Orchestrator Drivers

The first task an orchestrator driver must do is compile a list of all client devices that will be involved in a given transaction. For example, the NVMe Target driver creates a list including the namespace block device and the RNIC in use. If the orchestrator has access to a specific P2P provider to use it may check compatibility using `pci_p2pdma_distance()` otherwise it may find a memory provider that's compatible with all clients using `pci_p2pmem_find()`. If more than one provider is supported, the one nearest to all the clients will be chosen first. If more than one provider is an equal distance away, the one returned will be chosen at random (it is not an arbitrary but truly random). This function returns the PCI device to use for the provider with a reference taken and therefore when it's no longer needed it should be returned with `pci_dev_put()`.

Once a provider is selected, the orchestrator can then use `pci_alloc_p2pmem()` and `pci_free_p2pmem()` to allocate P2P memory from the provider. `pci_p2pmem_alloc_sgl()` and `pci_p2pmem_free_sgl()` are convenience functions for allocating scatter-gather lists with P2P memory.

Struct Page Caveats

Driver writers should be very careful about not passing these special struct pages to code that isn't prepared for it. At this time, the kernel interfaces do not have any checks for ensuring this. This obviously precludes passing these pages to userspace.

P2P memory is also technically IO memory but should never have any side effects behind it. Thus, the order of loads and stores should not be important and ioreadX(), iowriteX() and friends should not be necessary.

22.3.2 P2P DMA Support Library

int **pci_p2pdma_add_resource**(struct pci_dev * pdev, int bar, size_t size, u64 offset)
add memory for use as p2p memory

Parameters

struct pci_dev * pdev the device to add the memory to

int bar PCI BAR to add

size_t size size of the memory to add, may be zero to use the whole BAR

u64 offset offset into the PCI BAR

Description

The memory will be given ZONE_DEVICE struct pages so that it may be used with any DMA request.

int **pci_p2pdma_distance_many**(struct pci_dev * provider, struct device ** clients, int num_clients, bool verbose)
Determine the cumulative distance between a p2pdma provider and the clients in use.

Parameters

struct pci_dev * provider p2pdma provider to check against the client list

struct device ** clients array of devices to check (NULL-terminated)

int num_clients number of clients in the array

bool verbose if true, print warnings for devices when we return -1

Description

Returns -1 if any of the clients are not compatible, otherwise returns a positive number where a lower number is the preferable choice. (If there's one client that's the same as the provider it will return 0, which is best choice).

"compatible" means the provider and the clients are either all behind the same PCI root port or the host bridges connected to each of the devices are listed in the 'pci_p2pdma_whitelist'.

bool **pci_has_p2pmem**(struct pci_dev * pdev)
check if a given PCI device has published any p2pmem

Parameters

struct pci_dev * pdev PCI device to check

struct pci_dev * **pci_p2pmem_find_many**(struct device ** clients, int num_clients)
find a peer-to-peer DMA memory device compatible with the specified list of clients and shortest distance (as determined by pci_p2pmem_dma())

Parameters

struct device ** clients array of devices to check (NULL-terminated)

int num_clients number of client devices in the list

Description

If multiple devices are behind the same switch, the one "closest" to the client devices in use will be chosen first. (So if one of the providers is the same as one of the clients, that provider will be used ahead of any other providers that are unrelated). If multiple providers are an equal distance away, one will be chosen at random.

Returns a pointer to the PCI device with a reference taken (use pci_dev_put to return the reference) or NULL if no compatible device is found. The found provider will also be assigned to the client list.

void * **pci_alloc_p2pmem**(struct pci_dev * pdev, size_t size)
allocate peer-to-peer DMA memory

Parameters

struct pci_dev * pdev the device to allocate memory from

size_t size number of bytes to allocate

Description

Returns the allocated memory or NULL on error.

void **pci_free_p2pmem**(struct pci_dev * pdev, void * addr, size_t size)
free peer-to-peer DMA memory

Parameters

struct pci_dev * pdev the device the memory was allocated from

void * addr address of the memory that was allocated

size_t size number of bytes that were allocated

pci_bus_addr_t **pci_p2pmem_virt_to_bus**(struct pci_dev * pdev, void * addr)
return the PCI bus address for a given virtual address obtained with `pci_alloc_p2pmem()`

Parameters

struct pci_dev * pdev the device the memory was allocated from

void * addr address of the memory that was allocated

struct scatterlist * **pci_p2pmem_alloc_sgl**(struct pci_dev * pdev, unsigned int * nents, u32 length)
allocate peer-to-peer DMA memory in a scatterlist

Parameters

struct pci_dev * pdev the device to allocate memory from

unsigned int * nents the number of SG entries in the list

u32 length number of bytes to allocate

Return

`NULL` on error or `struct scatterlist` pointer and **nents** on success

void **pci_p2pmem_free_sgl**(struct pci_dev * pdev, struct scatterlist * sgl)
free a scatterlist allocated by `pci_p2pmem_alloc_sgl()`

Parameters

struct pci_dev * pdev the device to allocate memory from

struct scatterlist * sgl the allocated scatterlist

void **pci_p2pmem_publish**(struct pci_dev * pdev, bool publish)
publish the peer-to-peer DMA memory for use by other devices with pci_p2pmem_find()

Parameters

struct pci_dev * pdev the device with peer-to-peer DMA memory to publish

bool publish set to true to publish the memory, false to unpublish it

Description

Published memory can be used by other PCI device drivers for peer-2-peer DMA operations. Non-published memory is reserved for exclusive use of the device driver that registers the peer-to-peer memory.

int **pci_p2pdma_map_sg_attrs**(struct device * dev, struct scatterlist * sg, int nents, enum dma_data_direction dir, unsigned long attrs)
map a PCI peer-to-peer scatterlist for DMA

Parameters

struct device * dev device doing the DMA request

struct scatterlist * sg scatter list to map

int nents elements in the scatterlist

enum dma_data_direction dir DMA direction

unsigned long attrs DMA attributes passed to dma_map_sg() (if called)

Description

Scatterlists mapped with this function should be unmapped using `pci_p2pdma_unmap_sg_attrs()`.

Returns the number of SG entries mapped or 0 on error.

void **pci_p2pdma_unmap_sg_attrs**(struct device * dev, struct scatterlist * sg, int nents, enum dma_data_direction dir, unsigned long attrs)
unmap a PCI peer-to-peer scatterlist that was mapped with pci_p2pdma_map_sg()

Parameters

struct device * dev device doing the DMA request

struct scatterlist * sg scatter list to map

int nents number of elements returned by pci_p2pdma_map_sg()

enum dma_data_direction dir DMA direction

unsigned long attrs DMA attributes passed to dma_unmap_sg() (if called)

int **pci_p2pdma_enable_store**(const char * page, struct pci_dev ** p2p_dev, bool * use_p2pdma)
parse a configfs/sysfs attribute store to enable p2pdma

Parameters

const char * page contents of the value to be stored

struct pci_dev ** p2p_dev returns the PCI device that was selected to be used (if one was specified in the stored value)

bool * use_p2pdma returns whether to enable p2pdma or not

Description

Parses an attribute value to decide whether to enable p2pdma. The value can select a PCI device (using its full BDF device name) or a boolean (in any format strtobool() accepts). A false value disables p2pdma, a true value expects the caller to automatically find a compatible device and specifying a PCI device expects the caller to use the specific provider.

`pci_p2pdma_enable_show()` should be used as the show operation for the attribute.

Returns 0 on success

ssize_t **pci_p2pdma_enable_show**(char * page, struct pci_dev * p2p_dev, bool use_p2pdma)
show a configfs/sysfs attribute indicating whether p2pdma is enabled

Parameters

char * page contents of the stored value

struct pci_dev * p2p_dev the selected p2p device (NULL if no device is selected)

bool use_p2pdma whether p2pdma has been enabled

Description

Attributes that use `pci_p2pdma_enable_store()` should use this function to show the value of the attribute.

Returns 0 on success

CHAPTER

TWENTYTHREE

SERIAL PERIPHERAL INTERFACE (SPI)

SPI is the "Serial Peripheral Interface", widely used with embedded systems because it is a simple and efficient interface: basically a multiplexed shift register. Its three signal wires hold a clock (SCK, often in the range of 1-20 MHz), a "Master Out, Slave In" (MOSI) data line, and a "Master In, Slave Out" (MISO) data line. SPI is a full duplex protocol; for each bit shifted out the MOSI line (one per clock) another is shifted in on the MISO line. Those bits are assembled into words of various sizes on the way to and from system memory. An additional chipselect line is usually active-low (nCS); four signals are normally used for each peripheral, plus sometimes an interrupt.

The SPI bus facilities listed here provide a generalized interface to declare SPI busses and devices, manage them according to the standard Linux driver model, and perform input/output operations. At this time, only "master" side interfaces are supported, where Linux talks to SPI peripherals and does not implement such a peripheral itself. (Interfaces to support implementing SPI slaves would necessarily look different.)

The programming interface is structured around two kinds of driver, and two kinds of device. A "Controller Driver" abstracts the controller hardware, which may be as simple as a set of GPIO pins or as complex as a pair of FIFOs connected to dual DMA engines on the other side of the SPI shift register (maximizing throughput). Such drivers bridge between whatever bus they sit on (often the platform bus) and SPI, and expose the SPI side of their device as a `struct spi_master`. SPI devices are children of that master, represented as a `struct spi_device` and manufactured from `struct spi_board_info` descriptors which are usually provided by board-specific initialization code. A `struct spi_driver` is called a "Protocol Driver", and is bound to a spi_device using normal driver model calls.

The I/O model is a set of queued messages. Protocol drivers submit one or more `struct spi_message` objects, which are processed and completed asynchronously. (There are synchronous wrappers, however.) Messages are built from one or more `struct spi_transfer` objects, each of which wraps a full duplex SPI transfer. A variety of protocol tweaking options are needed, because different chips adopt very different policies for how they use the bits transferred with SPI.

struct **spi_statistics**
    statistics for spi transfers

Definition

```
struct spi_statistics {
  spinlock_t lock;
  unsigned long           messages;
  unsigned long           transfers;
  unsigned long           errors;
  unsigned long           timedout;
  unsigned long           spi_sync;
  unsigned long           spi_sync_immediate;
  unsigned long           spi_async;
  unsigned long long      bytes;
  unsigned long long      bytes_rx;
  unsigned long long      bytes_tx;
#define SPI_STATISTICS_HISTO_SIZE 17;
  unsigned long transfer_bytes_histo[SPI_STATISTICS_HISTO_SIZE];
  unsigned long transfers_split_maxsize;
};
```

Members

lock lock protecting this structure

messages number of spi-messages handled

transfers number of spi_transfers handled

errors number of errors during spi_transfer

timedout number of timeouts during spi_transfer

spi_sync number of times spi_sync is used

spi_sync_immediate number of times spi_sync is executed immediately in calling context without queuing and scheduling

spi_async number of times spi_async is used

bytes number of bytes transferred to/from device

bytes_rx number of bytes received from device

bytes_tx number of bytes sent to device

transfer_bytes_histo transfer bytes histogramm

transfers_split_maxsize number of transfers that have been split because of maxsize limit

struct **spi_delay**
SPI delay information

Definition

```
struct spi_delay {
#define SPI_DELAY_UNIT_USECS    0;
#define SPI_DELAY_UNIT_NSECS    1;
#define SPI_DELAY_UNIT_SCK      2;
  u16 value;
  u8 unit;
};
```

Members

value Value for the delay

unit Unit for the delay

struct **spi_device**
Controller side proxy for an SPI slave device

Definition

```
struct spi_device {
  struct device           dev;
  struct spi_controller   *controller;
  struct spi_controller   *master;
  u32 max_speed_hz;
  u8 chip_select;
  u8 bits_per_word;
  bool rt;
  u32 mode;
#define SPI_CPHA        0x01                    ;
#define SPI_CPOL        0x02                    ;
#define SPI_MODE_0      (0|0)                   ;
#define SPI_MODE_1      (0|SPI_CPHA);
#define SPI_MODE_2      (SPI_CPOL|0);
#define SPI_MODE_3      (SPI_CPOL|SPI_CPHA);
#define SPI_CS_HIGH     0x04                    ;
#define SPI_LSB_FIRST   0x08                    ;
#define SPI_3WIRE       0x10                    ;
#define SPI_LOOP        0x20                    ;
#define SPI_NO_CS       0x40                    ;
#define SPI_READY       0x80                    ;
#define SPI_TX_DUAL     0x100                   ;
#define SPI_TX_QUAD     0x200                   ;
#define SPI_RX_DUAL     0x400                   ;
#define SPI_RX_QUAD     0x800                   ;
#define SPI_CS_WORD     0x1000                  ;
#define SPI_TX_OCTAL    0x2000                  ;
#define SPI_RX_OCTAL    0x4000                  ;
#define SPI_3WIRE_HIZ   0x8000                  ;
  int irq;
  void *controller_state;
  void *controller_data;
  char modalias[SPI_NAME_SIZE];
  const char              *driver_override;
  int cs_gpio;
  struct gpio_desc        *cs_gpiod;
  struct spi_delay        word_delay;
  struct spi_statistics   statistics;
};
```

Members

dev Driver model representation of the device.

controller SPI controller used with the device.

master Copy of controller, for backwards compatibility.

max_speed_hz Maximum clock rate to be used with this chip (on this board); may be changed by the device's driver. The spi_transfer.speed_hz can override this for each transfer.

chip_select Chipselect, distinguishing chips handled by **controller**.

bits_per_word Data transfers involve one or more words; word sizes like eight or 12 bits are common. In-memory wordsizes are powers of two bytes (e.g. 20 bit samples use 32 bits). This may be changed by the device's driver, or left at the default (0) indicating protocol words are eight bit bytes. The spi_transfer.bits_per_word can override this for each transfer.

rt Make the pump thread real time priority.

mode The spi mode defines how data is clocked out and in. This may be changed by the device's driver. The "active low" default for chipselect mode can be overridden (by specifying SPI_CS_HIGH) as can the "MSB first" default for each word in a transfer (by specifying SPI_LSB_FIRST).

irq Negative, or the number passed to request_irq() to receive interrupts from this device.

controller_state Controller's runtime state

controller_data Board-specific definitions for controller, such as FIFO initialization parameters; from board_info.controller_data

modalias Name of the driver to use with this device, or an alias for that name. This appears in the sysfs "modalias" attribute for driver coldplugging, and in uevents used for hotplugging

driver_override If the name of a driver is written to this attribute, then the device will bind to the named driver and only the named driver.

cs_gpio LEGACY: gpio number of the chipselect line (optional, -ENOENT when not using a GPIO line) use cs_gpiod in new drivers by opting in on the spi_master.

cs_gpiod gpio descriptor of the chipselect line (optional, NULL when not using a GPIO line)

word_delay delay to be inserted between consecutive words of a transfer

statistics statistics for the spi_device

Description

A **spi_device** is used to interchange data between an SPI slave (usually a discrete chip) and CPU memory.

In **dev**, the platform_data is used to hold information about this device that's meaningful to the device's protocol driver, but not to its controller. One example might be an identifier for a chip variant with slightly different functionality; another might be information about how this particular board wires the chip's pins.

struct **spi_driver**
Host side "protocol" driver

Definition

```
struct spi_driver {
  const struct spi_device_id *id_table;
```

(continues on next page)

(continued from previous page)

```
  int (*probe)(struct spi_device *spi);
  int (*remove)(struct spi_device *spi);
  void (*shutdown)(struct spi_device *spi);
  struct device_driver    driver;
};
```

Members

id_table List of SPI devices supported by this driver

probe Binds this driver to the spi device. Drivers can verify that the device is actually present, and may need to configure characteristics (such as bits_per_word) which weren't needed for the initial configuration done during system setup.

remove Unbinds this driver from the spi device

shutdown Standard shutdown callback used during system state transitions such as powerdown/halt and kexec

driver SPI device drivers should initialize the name and owner field of this structure.

Description

This represents the kind of device driver that uses SPI messages to interact with the hardware at the other end of a SPI link. It's called a "protocol" driver because it works through messages rather than talking directly to SPI hardware (which is what the underlying SPI controller driver does to pass those messages). These protocols are defined in the specification for the device(s) supported by the driver.

As a rule, those device protocols represent the lowest level interface supported by a driver, and it will support upper level interfaces too. Examples of such upper levels include frameworks like MTD, networking, MMC, RTC, filesystem character device nodes, and hardware monitoring.

void **spi_unregister_driver**(struct spi_driver * sdrv)
reverse effect of spi_register_driver

Parameters

struct spi_driver * sdrv the driver to unregister

Context

can sleep

module_spi_driver(__spi_driver)
Helper macro for registering a SPI driver

Parameters

__spi_driver spi_driver struct

Description

Helper macro for SPI drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init()` and `module_exit()`

struct **spi_controller**
interface to SPI master or slave controller

Definition

```
struct spi_controller {
  struct device   dev;
  struct list_head list;
  s16 bus_num;
  u16 num_chipselect;
  u16 dma_alignment;
  u32 mode_bits;
  u32 buswidth_override_bits;
  u32 bits_per_word_mask;
#define SPI_BPW_MASK(bits) BIT((bits) - 1);
#define SPI_BPW_RANGE_MASK(min, max) GENMASK((max) - 1, (min) - 1);
  u32 min_speed_hz;
  u32 max_speed_hz;
  u16 flags;
#define SPI_CONTROLLER_HALF_DUPLEX      BIT(0)  ;
#define SPI_CONTROLLER_NO_RX            BIT(1)  ;
#define SPI_CONTROLLER_NO_TX            BIT(2)  ;
#define SPI_CONTROLLER_MUST_RX          BIT(3)  ;
#define SPI_CONTROLLER_MUST_TX          BIT(4)  ;
#define SPI_MASTER_GPIO_SS              BIT(5)  ;
  bool slave;
  size_t (*max_transfer_size)(struct spi_device *spi);
  size_t (*max_message_size)(struct spi_device *spi);
  struct mutex            io_mutex;
  spinlock_t bus_lock_spinlock;
  struct mutex            bus_lock_mutex;
  bool bus_lock_flag;
  int (*setup)(struct spi_device *spi);
  int (*set_cs_timing)(struct spi_device *spi, struct spi_delay *setup,␣
↪struct spi_delay *hold, struct spi_delay *inactive);
  int (*transfer)(struct spi_device *spi, struct spi_message *mesg);
  void (*cleanup)(struct spi_device *spi);
  bool (*can_dma)(struct spi_controller *ctlr,struct spi_device *spi,␣
↪struct spi_transfer *xfer);
  bool queued;
  struct kthread_worker           kworker;
  struct task_struct              *kworker_task;
  struct kthread_work             pump_messages;
  spinlock_t queue_lock;
  struct list_head                queue;
  struct spi_message              *cur_msg;
  bool idling;
  bool busy;
  bool running;
  bool rt;
  bool auto_runtime_pm;
  bool cur_msg_prepared;
  bool cur_msg_mapped;
  struct completion               xfer_completion;
  size_t max_dma_len;
  int (*prepare_transfer_hardware)(struct spi_controller *ctlr);
  int (*transfer_one_message)(struct spi_controller *ctlr, struct spi_
↪message *mesg);
```

(continues on next page)

(continued from previous page)

```
  int (*unprepare_transfer_hardware)(struct spi_controller *ctlr);
  int (*prepare_message)(struct spi_controller *ctlr, struct spi_message␣
↪*message);
  int (*unprepare_message)(struct spi_controller *ctlr, struct spi_message␣
↪*message);
  int (*slave_abort)(struct spi_controller *ctlr);
  void (*set_cs)(struct spi_device *spi, bool enable);
  int (*transfer_one)(struct spi_controller *ctlr, struct spi_device *spi,␣
↪struct spi_transfer *transfer);
  void (*handle_err)(struct spi_controller *ctlr, struct spi_message␣
↪*message);
  const struct spi_controller_mem_ops *mem_ops;
  struct spi_delay        cs_setup;
  struct spi_delay        cs_hold;
  struct spi_delay        cs_inactive;
  int *cs_gpios;
  struct gpio_desc        **cs_gpiods;
  bool use_gpio_descriptors;
  u8 unused_native_cs;
  u8 max_native_cs;
  struct spi_statistics   statistics;
  struct dma_chan         *dma_tx;
  struct dma_chan         *dma_rx;
  void *dummy_rx;
  void *dummy_tx;
  int (*fw_translate_cs)(struct spi_controller *ctlr, unsigned cs);
  bool ptp_sts_supported;
  unsigned long           irq_flags;
};
```

Members

dev device interface to this driver

list link with the global spi_controller list

bus_num board-specific (and often SOC-specific) identifier for a given SPI controller.

num_chipselect chipselects are used to distinguish individual SPI slaves, and are numbered from zero to num_chipselects. each slave has a chipselect signal, but it' s common that not every chipselect is connected to a slave.

dma_alignment SPI controller constraint on DMA buffers alignment.

mode_bits flags understood by this controller driver

bits_per_word_mask A mask indicating which values of bits_per_word are supported by the driver. Bit n indicates that a bits_per_word n+1 is supported. If set, the SPI core will reject any transfer with an unsupported bits_per_word. If not set, this value is simply ignored, and it' s up to the individual driver to perform any validation.

min_speed_hz Lowest supported transfer speed

max_speed_hz Highest supported transfer speed

flags other constraints relevant to this driver

slave indicates that this is an SPI slave controller

max_transfer_size function that returns the max transfer size for a `spi_device`; may be `NULL`, so the default `SIZE_MAX` will be used.

max_message_size function that returns the max message size for a `spi_device`; may be `NULL`, so the default `SIZE_MAX` will be used.

io_mutex mutex for physical bus access

bus_lock_spinlock spinlock for SPI bus locking

bus_lock_mutex mutex for exclusion of multiple callers

bus_lock_flag indicates that the SPI bus is locked for exclusive use

setup updates the device mode and clocking records used by a device's SPI controller; protocol code may call this. This must fail if an unrecognized or unsupported mode is requested. It's always safe to call this unless transfers are pending on the device whose settings are being modified.

set_cs_timing optional hook for SPI devices to request SPI master controller for configuring specific CS setup time, hold time and inactive delay interms of clock counts

transfer adds a message to the controller's transfer queue.

cleanup frees controller-specific state

can_dma determine whether this controller supports DMA

queued whether this controller is providing an internal message queue

kworker thread struct for message pump

kworker_task pointer to task for message pump kworker thread

pump_messages work struct for scheduling work to the message pump

queue_lock spinlock to syncronise access to message queue

queue message queue

cur_msg the currently in-flight message

idling the device is entering idle state

busy message pump is busy

running message pump is running

rt whether this queue is set to run as a realtime task

auto_runtime_pm the core should ensure a runtime PM reference is held while the hardware is prepared, using the parent device for the spidev

cur_msg_prepared spi_prepare_message was called for the currently in-flight message

cur_msg_mapped message has been mapped for DMA

xfer_completion used by core transfer_one_message()

max_dma_len Maximum length of a DMA transfer for the device.

prepare_transfer_hardware a message will soon arrive from the queue so the subsystem requests the driver to prepare the transfer hardware by issuing this call

transfer_one_message the subsystem calls the driver to transfer a single message while queuing transfers that arrive in the meantime. When the driver is finished with this message, it must call `spi_finalize_current_message()` so the subsystem can issue the next message

unprepare_transfer_hardware there are currently no more messages on the queue so the subsystem notifies the driver that it may relax the hardware by issuing this call

prepare_message set up the controller to transfer a single message, for example doing DMA mapping. Called from threaded context.

unprepare_message undo any work done by prepare_message().

slave_abort abort the ongoing transfer request on an SPI slave controller

set_cs set the logic level of the chip select line. May be called from interrupt context.

transfer_one transfer a single spi_transfer.

- return 0 if the transfer is finished,
- return 1 if the transfer is still in progress. When the driver is finished with this transfer it must call `spi_finalize_current_transfer()` so the subsystem can issue the next transfer. Note: transfer_one and transfer_one_message are mutually exclusive; when both are set, the generic subsystem does not call your transfer_one callback.

handle_err the subsystem calls the driver to handle an error that occurs in the generic implementation of transfer_one_message().

mem_ops optimized/dedicated operations for interactions with SPI memory. This field is optional and should only be implemented if the controller has native support for memory like operations.

cs_setup delay to be introduced by the controller after CS is asserted

cs_hold delay to be introduced by the controller before CS is deasserted

cs_inactive delay to be introduced by the controller after CS is deasserted. If **cs_change_delay** is used from **spi_transfer**, then the two delays will be added up.

cs_gpios LEGACY: array of GPIO descs to use as chip select lines; one per CS number. Any individual value may be -ENOENT for CS lines that are not GPIOs (driven by the SPI controller itself). Use the cs_gpiods in new drivers.

cs_gpiods Array of GPIO descs to use as chip select lines; one per CS number. Any individual value may be NULL for CS lines that are not GPIOs (driven by the SPI controller itself).

use_gpio_descriptors Turns on the code in the SPI core to parse and grab GPIO descriptors rather than using global GPIO numbers grabbed by the driver. This will fill in **cs_gpiods** and **cs_gpios** should not be used, and SPI devices will have the cs_gpiod assigned rather than cs_gpio.

unused_native_cs When cs_gpiods is used, `spi_register_controller()` will fill in this field with the first unused native CS, to be used by SPI controller drivers that need to drive a native CS when using GPIO CS.

max_native_cs When cs_gpiods is used, and this field is filled in, `spi_register_controller()` will validate all native CS (including the unused native CS) against this value.

statistics statistics for the spi_controller

dma_tx DMA transmit channel

dma_rx DMA receive channel

dummy_rx dummy receive buffer for full-duplex devices

dummy_tx dummy transmit buffer for full-duplex devices

fw_translate_cs If the boot firmware uses different numbering scheme what Linux expects, this optional hook can be used to translate between the two.

ptp_sts_supported If the driver sets this to true, it must provide a time snapshot in **spi_transfer->ptp_sts** as close as possible to the moment in time when **spi_transfer->ptp_sts_word_pre** and **spi_transfer->ptp_sts_word_post** were transmitted. If the driver does not set this, the SPI core takes the snapshot as close to the driver hand-over as possible.

irq_flags Interrupt enable state during PTP system timestamping

Description

Each SPI controller can communicate with one or more **spi_device** children. These make a small bus, sharing MOSI, MISO and SCK signals but not chip select signals. Each device may be configured to use a different clock rate, since those shared signals are ignored unless the chip is selected.

The driver for an SPI controller manages access to those devices through a queue of spi_message transactions, copying data between CPU memory and an SPI slave device. For each such message it queues, it calls the message' s completion function when the transaction completes.

struct **spi_res**
spi resource management structure

Definition

```
struct spi_res {
  struct list_head        entry;
  spi_res_release_t release;
  unsigned long long      data[];
};
```

Members

entry list entry

release release code called prior to freeing this resource

data extra data allocated for the specific use-case

Description

this is based on ideas from devres, but focused on life-cycle management during spi_message processing

struct **spi_transfer**
a read/write buffer pair

Definition

```
struct spi_transfer {
  const void      *tx_buf;
  void *rx_buf;
  unsigned len;
  dma_addr_t tx_dma;
  dma_addr_t rx_dma;
  struct sg_table tx_sg;
  struct sg_table rx_sg;
  unsigned cs_change:1;
  unsigned tx_nbits:3;
  unsigned rx_nbits:3;
#define SPI_NBITS_SINGLE        0x01 ;
#define SPI_NBITS_DUAL          0x02 ;
#define SPI_NBITS_QUAD          0x04 ;
  u8 bits_per_word;
  u16 delay_usecs;
  struct spi_delay        delay;
  struct spi_delay        cs_change_delay;
  struct spi_delay        word_delay;
  u32 speed_hz;
  u32 effective_speed_hz;
  unsigned int    ptp_sts_word_pre;
  unsigned int    ptp_sts_word_post;
  struct ptp_system_timestamp *ptp_sts;
  bool timestamped;
  struct list_head transfer_list;
};
```

Members

tx_buf data to be written (dma-safe memory), or NULL

rx_buf data to be read (dma-safe memory), or NULL

len size of rx and tx buffers (in bytes)

tx_dma DMA address of tx_buf, if **spi_message.is_dma_mapped**

rx_dma DMA address of rx_buf, if **spi_message.is_dma_mapped**

tx_sg Scatterlist for transmit, currently not for client use

rx_sg Scatterlist for receive, currently not for client use

cs_change affects chipselect after this transfer completes

tx_nbits number of bits used for writing. If 0 the default (SPI_NBITS_SINGLE) is used.

rx_nbits number of bits used for reading. If 0 the default (SPI_NBITS_SINGLE) is used.

bits_per_word select a bits_per_word other than the device default for this transfer. If 0 the default (from **spi_device**) is used.

delay_usecs microseconds to delay after this transfer before (optionally) changing the chipselect status, then starting the next transfer or completing this **spi_message**.

delay delay to be introduced after this transfer before (optionally) changing the chipselect status, then starting the next transfer or completing this **spi_message**.

cs_change_delay delay between cs deassert and assert when **cs_change** is set and **spi_transfer** is not the last in **spi_message**

word_delay inter word delay to be introduced after each word size (set by bits_per_word) transmission.

speed_hz Select a speed other than the device default for this transfer. If 0 the default (from **spi_device**) is used.

effective_speed_hz the effective SCK-speed that was used to transfer this transfer. Set to 0 if the spi bus driver does not support it.

ptp_sts_word_pre The word (subject to bits_per_word semantics) offset within **tx_buf** for which the SPI device is requesting that the time snapshot for this transfer begins. Upon completing the SPI transfer, this value may have changed compared to what was requested, depending on the available snapshotting resolution (DMA transfer, **ptp_sts_supported** is false, etc).

ptp_sts_word_post See **ptp_sts_word_post**. The two can be equal (meaning that a single byte should be snapshotted). If the core takes care of the timestamp (if **ptp_sts_supported** is false for this controller), it will set **ptp_sts_word_pre** to 0, and **ptp_sts_word_post** to the length of the transfer. This is done purposefully (instead of setting to spi_transfer->len - 1) to denote that a transfer-level snapshot taken from within the driver may still be of higher quality.

ptp_sts Pointer to a memory location held by the SPI slave device where a PTP system timestamp structure may lie. If drivers use PIO or their hardware has some sort of assist for retrieving exact transfer timing, they can (and should) assert **ptp_sts_supported** and populate this structure using the ptp_read_system_*ts helper functions. The timestamp must represent the time at which the SPI slave device has processed the word, i.e. the "pre" timestamp should be taken before transmitting the "pre" word, and the "post" timestamp after receiving transmit confirmation from the controller for the "post" word.

transfer_list transfers are sequenced through **spi_message.transfers**

Description

SPI transfers always write the same number of bytes as they read. Protocol drivers should always provide **rx_buf** and/or **tx_buf**. In some cases, they may also want to provide DMA addresses for the data being transferred; that may reduce overhead, when the underlying driver uses dma.

If the transmit buffer is null, zeroes will be shifted out while filling **rx_buf**. If the receive buffer is null, the data shifted in will be discarded. Only "len" bytes shift

out (or in). It's an error to try to shift out a partial word. (For example, by shifting out three bytes with word size of sixteen or twenty bits; the former uses two bytes per word, the latter uses four bytes.)

In-memory data values are always in native CPU byte order, translated from the wire byte order (big-endian except with SPI_LSB_FIRST). So for example when bits_per_word is sixteen, buffers are 2N bytes long (**len** = 2N) and hold N sixteen bit words in CPU byte order.

When the word size of the SPI transfer is not a power-of-two multiple of eight bits, those in-memory words include extra bits. In-memory words are always seen by protocol drivers as right-justified, so the undefined (rx) or unused (tx) bits are always the most significant bits.

All SPI transfers start with the relevant chipselect active. Normally it stays selected until after the last transfer in a message. Drivers can affect the chipselect signal using cs_change.

(i) If the transfer isn't the last one in the message, this flag is used to make the chipselect briefly go inactive in the middle of the message. Toggling chipselect in this way may be needed to terminate a chip command, letting a single spi_message perform all of group of chip transactions together.

(ii) When the transfer is the last one in the message, the chip may stay selected until the next transfer. On multi-device SPI busses with nothing blocking messages going to other devices, this is just a performance hint; starting a message to another device deselects this one. But in other cases, this can be used to ensure correctness. Some devices need protocol transactions to be built from a series of spi_message submissions, where the content of one message is determined by the results of previous messages and where the whole transaction ends when the chipselect goes intactive.

When SPI can transfer in 1x,2x or 4x. It can get this transfer information from device through **tx_nbits** and **rx_nbits**. In Bi-direction, these two should both be set. User can set transfer mode with SPI_NBITS_SINGLE(1x) SPI_NBITS_DUAL(2x) and SPI_NBITS_QUAD(4x) to support these three transfer.

The code that submits an spi_message (and its spi_transfers) to the lower layers is responsible for managing its memory. Zero-initialize every field you don't set up explicitly, to insulate against future API updates. After you submit a message and its transfers, ignore them until its completion callback.

struct **spi_message**
one multi-segment SPI transaction

Definition

```
struct spi_message {
  struct list_head        transfers;
  struct spi_device       *spi;
  unsigned is_dma_mapped:1;
  void (*complete)(void *context);
  void *context;
  unsigned frame_length;
  unsigned actual_length;
  int status;
```

(continues on next page)

(continued from previous page)

```
  struct list_head        queue;
  void *state;
  struct list_head        resources;
};
```

Members

transfers list of transfer segments in this transaction

spi SPI device to which the transaction is queued

is_dma_mapped if true, the caller provided both dma and cpu virtual addresses for each transfer buffer

complete called to report transaction completions

context the argument to complete() when it's called

frame_length the total number of bytes in the message

actual_length the total number of bytes that were transferred in all successful segments

status zero for success, else negative errno

queue for use by whichever driver currently owns the message

state for use by whichever driver currently owns the message

resources for resource management when the spi message is processed

Description

A **spi_message** is used to execute an atomic sequence of data transfers, each represented by a struct spi_transfer. The sequence is "atomic" in the sense that no other spi_message may use that SPI bus until that sequence completes. On some systems, many such sequences can execute as as single programmed DMA transfer. On all systems, these messages are queued, and might complete after transactions to other devices. Messages sent to a given spi_device are always executed in FIFO order.

The code that submits an spi_message (and its spi_transfers) to the lower layers is responsible for managing its memory. Zero-initialize every field you don't set up explicitly, to insulate against future API updates. After you submit a message and its transfers, ignore them until its completion callback.

void **spi_message_init_with_transfers**(struct spi_message * m, struct spi_transfer * xfers, unsigned int num_xfers)

Initialize spi_message and append transfers

Parameters

struct spi_message * m spi_message to be initialized

struct spi_transfer * xfers An array of spi transfers

unsigned int num_xfers Number of items in the xfer array

Description

This function initializes the given spi_message and adds each spi_transfer in the given array to the message.

bool **spi_is_bpw_supported**(struct spi_device * spi, u32 bpw)
Check if bits per word is supported

Parameters

struct spi_device * spi SPI device

u32 bpw Bits per word

Description

This function checks to see if the SPI controller supports **bpw**.

Return

True if **bpw** is supported, false otherwise.

struct **spi_replaced_transfers**
structure describing the spi_transfer replacements that have occurred so that they can get reverted

Definition

```
struct spi_replaced_transfers {
  spi_replaced_release_t release;
  void *extradata;
  struct list_head replaced_transfers;
  struct list_head *replaced_after;
  size_t inserted;
  struct spi_transfer inserted_transfers[];
};
```

Members

release some extra release code to get executed prior to relasing this structure

extradata pointer to some extra data if requested or NULL

replaced_transfers transfers that have been replaced and which need to get restored

replaced_after the transfer after which the **replaced_transfers** are to get re-inserted

inserted number of transfers inserted

inserted_transfers array of spi_transfers of array-size **inserted**, that have been replacing replaced_transfers

note

that **extradata** will point to **inserted_transfers**[**inserted**] if some extra allocation is requested, so alignment will be the same as for spi_transfers

int **spi_sync_transfer**(struct spi_device * spi, struct spi_transfer * xfers, unsigned int num_xfers)
synchronous SPI data transfer

Parameters

struct spi_device * spi device with which data will be exchanged

struct spi_transfer * xfers An array of spi_transfers

unsigned int num_xfers Number of items in the xfer array

Context

can sleep

Description

Does a synchronous SPI data transfer of the given spi_transfer array.

For more specific semantics see `spi_sync()`.

Return

Return: zero on success, else a negative error code.

int **spi_write**(struct spi_device * spi, const void * buf, size_t len)
SPI synchronous write

Parameters

struct spi_device * spi device to which data will be written

const void * buf data buffer

size_t len data buffer size

Context

can sleep

Description

This function writes the buffer **buf**. Callable only from contexts that can sleep.

Return

zero on success, else a negative error code.

int **spi_read**(struct spi_device * spi, void * buf, size_t len)
SPI synchronous read

Parameters

struct spi_device * spi device from which data will be read

void * buf data buffer

size_t len data buffer size

Context

can sleep

Description

This function reads the buffer **buf**. Callable only from contexts that can sleep.

Return

zero on success, else a negative error code.

ssize_t **spi_w8r8**(struct spi_device * spi, u8 cmd)
SPI synchronous 8 bit write followed by 8 bit read

Parameters

struct spi_device * spi device with which data will be exchanged

u8 cmd command to be written before data is read back

Context

can sleep

Description

Callable only from contexts that can sleep.

Return

the (unsigned) eight bit number returned by the device, or else a negative error code.

ssize_t **spi_w8r16**(struct spi_device * spi, u8 cmd)
SPI synchronous 8 bit write followed by 16 bit read

Parameters

struct spi_device * spi device with which data will be exchanged

u8 cmd command to be written before data is read back

Context

can sleep

Description

The number is returned in wire-order, which is at least sometimes big-endian.

Callable only from contexts that can sleep.

Return

the (unsigned) sixteen bit number returned by the device, or else a negative error code.

ssize_t **spi_w8r16be**(struct spi_device * spi, u8 cmd)
SPI synchronous 8 bit write followed by 16 bit big-endian read

Parameters

struct spi_device * spi device with which data will be exchanged

u8 cmd command to be written before data is read back

Context

can sleep

Description

This function is similar to spi_w8r16, with the exception that it will convert the read 16 bit data word from big-endian to native endianness.

Callable only from contexts that can sleep.

Return

the (unsigned) sixteen bit number returned by the device in cpu endianness, or else a negative error code.

struct **spi_board_info**
board-specific template for a SPI device

Definition

```
struct spi_board_info {
  char modalias[SPI_NAME_SIZE];
  const void      *platform_data;
  const struct property_entry *properties;
  void *controller_data;
  int irq;
  u32 max_speed_hz;
  u16 bus_num;
  u16 chip_select;
  u32 mode;
};
```

Members

modalias Initializes spi_device.modalias; identifies the driver.

platform_data Initializes spi_device.platform_data; the particular data stored there is driver-specific.

properties Additional device properties for the device.

controller_data Initializes spi_device.controller_data; some controllers need hints about hardware setup, e.g. for DMA.

irq Initializes spi_device.irq; depends on how the board is wired.

max_speed_hz Initializes spi_device.max_speed_hz; based on limits from the chip datasheet and board-specific signal quality issues.

bus_num Identifies which spi_controller parents the spi_device; unused by `spi_new_device()`, and otherwise depends on board wiring.

chip_select Initializes spi_device.chip_select; depends on how the board is wired.

mode Initializes spi_device.mode; based on the chip datasheet, board wiring (some devices support both 3WIRE and standard modes), and possibly presence of an inverter in the chipselect path.

Description

When adding new SPI devices to the device tree, these structures serve as a partial device template. They hold information which can't always be determined by drivers. Information that probe() can establish (such as the default transfer wordsize) is not included here.

These structures are used in two places. Their primary role is to be stored in tables of board-specific device descriptors, which are declared early in board initialization and then used (much later) to populate a controller's device tree after the that controller's driver initializes. A secondary (and atypical) role is as a parameter

to `spi_new_device()` call, which happens after those controller drivers are active in some dynamic board configuration models.

int **spi_register_board_info**(struct spi_board_info const * info, unsigned n)
register SPI devices for a given board

Parameters

struct spi_board_info const * info array of chip descriptors

unsigned n how many descriptors are provided

Context

can sleep

Description

Board-specific early init code calls this (probably during arch_initcall) with segments of the SPI device table. Any device nodes are created later, after the relevant parent SPI controller (bus_num) is defined. We keep this table of devices forever, so that reloading a controller driver will not make Linux forget about these hard-wired devices.

Other code can also call this, e.g. a particular add-on board might provide SPI devices through its expansion connector, so code initializing that board would naturally declare its SPI devices.

The board info passed can safely be __initdata …but be careful of any embedded pointers (platform_data, etc), they' re copied as-is. Device properties are deep-copied though.

Return

zero on success, else a negative error code.

int **__spi_register_driver**(struct module * owner, struct spi_driver * sdrv)
register a SPI driver

Parameters

struct module * owner owner module of the driver to register

struct spi_driver * sdrv the driver to register

Context

can sleep

Return

zero on success, else a negative error code.

struct spi_device * **spi_alloc_device**(struct spi_controller * ctlr)
Allocate a new SPI device

Parameters

struct spi_controller * ctlr Controller to which device is connected

Context

can sleep

Description

Allows a driver to allocate and initialize a spi_device without registering it immediately. This allows a driver to directly fill the spi_device with device parameters before calling `spi_add_device()` on it.

Caller is responsible to call `spi_add_device()` on the returned spi_device structure to add it to the SPI controller. If the caller needs to discard the spi_device without adding it, then it should call spi_dev_put() on it.

Return

a pointer to the new device, or NULL.

int **spi_add_device**(struct spi_device * spi)
Add spi_device allocated with spi_alloc_device

Parameters

struct spi_device * spi spi_device to register

Description

Companion function to spi_alloc_device. Devices allocated with spi_alloc_device can be added onto the spi bus with this function.

Return

0 on success; negative errno on failure

struct spi_device * **spi_new_device**(struct spi_controller * ctlr, struct spi_board_info * chip)
instantiate one new SPI device

Parameters

struct spi_controller * ctlr Controller to which device is connected

struct spi_board_info * chip Describes the SPI device

Context

can sleep

Description

On typical mainboards, this is purely internal; and it's not needed after board init creates the hard-wired devices. Some development platforms may not be able to use spi_register_board_info though, and this is exported so that for example a USB or parport based adapter driver could add devices (which it would learn about out-of-band).

Return

the new device, or NULL.

void **spi_unregister_device**(struct spi_device * spi)
unregister a single SPI device

Parameters

struct spi_device * spi spi_device to unregister

Description

Start making the passed SPI device vanish. Normally this would be handled by `spi_unregister_controller()`.

void **spi_finalize_current_transfer**(struct spi_controller * ctlr)
report completion of a transfer

Parameters

struct spi_controller * ctlr the controller reporting completion

Description

Called by SPI drivers using the core transfer_one_message() implementation to notify it that the current interrupt driven transfer has finished and the next one may be scheduled.

void **spi_take_timestamp_pre**(struct spi_controller * ctlr, struct spi_transfer * xfer, size_t progress, bool irqs_off)
helper for drivers to collect the beginning of the TX timestamp for the requested byte from the SPI transfer. The frequency with which this function must be called (once per word, once for the whole transfer, once per batch of words etc) is arbitrary as long as the **tx** buffer offset is greater than or equal to the requested byte at the time of the call. The timestamp is only taken once, at the first such call. It is assumed that the driver advances its **tx** buffer pointer monotonically.

Parameters

struct spi_controller * ctlr Pointer to the spi_controller structure of the driver

struct spi_transfer * xfer Pointer to the transfer being timestamped

size_t progress How many words (not bytes) have been transferred so far

bool irqs_off If true, will disable IRQs and preemption for the duration of the transfer, for less jitter in time measurement. Only compatible with PIO drivers. If true, must follow up with spi_take_timestamp_post or otherwise system will crash. WARNING: for fully predictable results, the CPU frequency must also be under control (governor).

void **spi_take_timestamp_post**(struct spi_controller * ctlr, struct spi_transfer * xfer, size_t progress, bool irqs_off)
helper for drivers to collect the end of the TX timestamp for the requested byte from the SPI transfer. Can be called with an arbitrary frequency: only the first call where **tx** exceeds or is equal to the requested word will be timestamped.

Parameters

struct spi_controller * ctlr Pointer to the spi_controller structure of the driver

struct spi_transfer * xfer Pointer to the transfer being timestamped

size_t progress How many words (not bytes) have been transferred so far

bool irqs_off If true, will re-enable IRQs and preemption for the local CPU.

struct spi_message * **spi_get_next_queued_message**(struct spi_controller * ctlr)
called by driver to check for queued messages

Parameters

struct spi_controller * ctlr the controller to check for queued messages

Description

If there are more messages in the queue, the next message is returned from this call.

Return

the next message in the queue, else NULL if the queue is empty.

void **spi_finalize_current_message**(struct spi_controller * ctlr)
the current message is complete

Parameters

struct spi_controller * ctlr the controller to return the message to

Description

Called by the driver to notify the core that the message in the front of the queue is complete and can be removed from the queue.

int **spi_slave_abort**(struct spi_device * spi)
abort the ongoing transfer request on an SPI slave controller

Parameters

struct spi_device * spi device used for the current transfer

struct spi_controller * **__spi_alloc_controller**(struct device * dev, unsigned int size, bool slave)
allocate an SPI master or slave controller

Parameters

struct device * dev the controller, possibly using the platform_bus

unsigned int size how much zeroed driver-private data to allocate; the pointer to this memory is in the driver_data field of the returned device, accessible with spi_controller_get_devdata(); the memory is cacheline aligned; drivers granting DMA access to portions of their private data need to round up **size** using ALIGN(size, dma_get_cache_alignment()).

bool slave flag indicating whether to allocate an SPI master (false) or SPI slave (true) controller

Context

can sleep

Description

This call is used only by SPI controller drivers, which are the only ones directly touching chip registers. It's how they allocate an spi_controller structure, prior to calling `spi_register_controller()`.

This must be called from context that can sleep.

The caller is responsible for assigning the bus number and initializing the controller's methods before calling `spi_register_controller()`; and (after errors adding the device) calling spi_controller_put() to prevent a memory leak.

Return

the SPI controller structure on success, else NULL.

int **spi_register_controller**(struct spi_controller * ctlr)
register SPI master or slave controller

Parameters

struct spi_controller * ctlr initialized master, originally from spi_alloc_master() or spi_alloc_slave()

Context

can sleep

Description

SPI controllers connect to their drivers using some non-SPI bus, such as the platform bus. The final stage of probe() in that code includes calling `spi_register_controller()` to hook up to this SPI bus glue.

SPI controllers use board specific (often SOC specific) bus numbers, and board-specific addressing for SPI devices combines those numbers with chip select numbers. Since SPI does not directly support dynamic device identification, boards need configuration tables telling which chip is at which address.

This must be called from context that can sleep. It returns zero on success, else a negative error code (dropping the controller's refcount). After a successful return, the caller is responsible for calling `spi_unregister_controller()`.

Return

zero on success, else a negative error code.

int **devm_spi_register_controller**(struct device * dev, struct spi_controller * ctlr)
register managed SPI master or slave controller

Parameters

struct device * dev device managing SPI controller

struct spi_controller * ctlr initialized controller, originally from spi_alloc_master() or spi_alloc_slave()

Context

can sleep

Description

Register a SPI device as with `spi_register_controller()` which will automatically be unregistered and freed.

Return

zero on success, else a negative error code.

void **spi_unregister_controller**(struct spi_controller * ctlr)
unregister SPI master or slave controller

Parameters

struct spi_controller * ctlr the controller being unregistered

Context

can sleep

Description

This call is used only by SPI controller drivers, which are the only ones directly touching chip registers.

This must be called from context that can sleep.

Note that this function also drops a reference to the controller.

struct spi_controller * **spi_busnum_to_master**(u16 bus_num)
look up master associated with bus_num

Parameters

u16 bus_num the master's bus number

Context

can sleep

Description

This call may be used with devices that are registered after arch init time. It returns a refcounted pointer to the relevant spi_controller (which the caller must release), or NULL if there is no such master registered.

Return

the SPI master structure on success, else NULL.

void * **spi_res_alloc**(struct spi_device * spi, spi_res_release_t release, size_t size, gfp_t gfp)
allocate a spi resource that is life-cycle managed during the processing of a spi_message while using spi_transfer_one

Parameters

struct spi_device * spi the spi device for which we allocate memory

spi_res_release_t release the release code to execute for this resource

size_t size size to alloc and return

gfp_t gfp GFP allocation flags

Return

the pointer to the allocated data

Description

This may get enhanced in the future to allocate from a memory pool of the **spi_device** or **spi_controller** to avoid repeated allocations.

void **spi_res_free**(void * res)
free an spi resource

Parameters

void * res pointer to the custom data of a resource

void **spi_res_add**(struct spi_message * message, void * res)
add a spi_res to the spi_message

Parameters

struct spi_message * message the spi message

void * res the spi_resource

void **spi_res_release**(struct spi_controller * ctlr, struct spi_message * message)
release all spi resources for this message

Parameters

struct spi_controller * ctlr the **spi_controller**

struct spi_message * message the **spi_message**

struct spi_replaced_transfers * **spi_replace_transfers**(struct spi_message * msg, struct spi_transfer * xfer_first, size_t remove, size_t insert, spi_replaced_release_t release, size_t extradatasize, gfp_t gfp)
replace transfers with several transfers and register change with spi_message.resources

Parameters

struct spi_message * msg the spi_message we work upon

struct spi_transfer * xfer_first the first spi_transfer we want to replace

size_t remove number of transfers to remove

size_t insert the number of transfers we want to insert instead

spi_replaced_release_t release extra release code necessary in some circumstances

size_t extradatasize extra data to allocate (with alignment guarantees of struct **spi_transfer**)

gfp_t gfp gfp flags

Return

pointer to spi_replaced_transfers, PTR_ERR(…) in case of errors.

int **spi_split_transfers_maxsize**(struct spi_controller * ctlr, struct spi_message * msg, size_t maxsize, gfp_t gfp)
split spi transfers into multiple transfers when an individual transfer exceeds a certain size

Parameters

struct spi_controller * ctlr the **spi_controller** for this transfer

struct spi_message * msg the **spi_message** to transform

size_t maxsize the maximum when to apply this

gfp_t gfp GFP allocation flags

Return

status of transformation

int **spi_setup**(struct spi_device * spi)
setup SPI mode and clock rate

Parameters

struct spi_device * spi the device whose settings are being modified

Context

can sleep, and no requests are queued to the device

Description

SPI protocol drivers may need to update the transfer mode if the device doesn't work with its default. They may likewise need to update clock rates or word sizes from initial values. This function changes those settings, and must be called from a context that can sleep. Except for SPI_CS_HIGH, which takes effect immediately, the changes take effect the next time the device is selected and data is transferred to or from it. When this function returns, the spi device is deselected.

Note that this call will fail if the protocol driver specifies an option that the underlying controller or its driver does not support. For example, not all hardware supports wire transfers using nine bit words, LSB-first wire encoding, or active-high chipselects.

Return

zero on success, else a negative error code.

int **spi_set_cs_timing**(struct spi_device * spi, struct spi_delay * setup, struct spi_delay * hold, struct spi_delay * inactive)
configure CS setup, hold, and inactive delays

Parameters

struct spi_device * spi the device that requires specific CS timing configuration

struct spi_delay * setup CS setup time specified via **spi_delay**

struct spi_delay * hold CS hold time specified via **spi_delay**

struct spi_delay * inactive CS inactive delay between transfers specified via **spi_delay**

Return

zero on success, else a negative error code.

int **spi_async**(struct spi_device * spi, struct spi_message * message)
asynchronous SPI transfer

Parameters

struct spi_device * spi device with which data will be exchanged

struct spi_message * message describes the data transfers, including completion callback

Context

any (irqs may be blocked, etc)

Description

This call may be used in_irq and other contexts which can't sleep, as well as from task contexts which can sleep.

The completion callback is invoked in a context which can't sleep. Before that invocation, the value of message->status is undefined. When the callback is issued, message->status holds either zero (to indicate complete success) or a negative error code. After that callback returns, the driver which issued the transfer request may deallocate the associated memory; it's no longer in use by any SPI core or controller driver code.

Note that although all messages to a spi_device are handled in FIFO order, messages may go to different devices in other orders. Some device might be higher priority, or have various "hard" access time requirements, for example.

On detection of any fault during the transfer, processing of the entire message is aborted, and the device is deselected. Until returning from the associated message completion callback, no other spi_message queued to that device will be processed. (This rule applies equally to all the synchronous transfer calls, which are wrappers around this core asynchronous primitive.)

Return

zero on success, else a negative error code.

int **spi_async_locked**(struct spi_device * spi, struct spi_message * message)
version of spi_async with exclusive bus usage

Parameters

struct spi_device * spi device with which data will be exchanged

struct spi_message * message describes the data transfers, including completion callback

Context

any (irqs may be blocked, etc)

Description

This call may be used in_irq and other contexts which can't sleep, as well as from task contexts which can sleep.

The completion callback is invoked in a context which can't sleep. Before that invocation, the value of message->status is undefined. When the callback is issued, message->status holds either zero (to indicate complete success) or a negative error code. After that callback returns, the driver which issued the transfer request may deallocate the associated memory; it's no longer in use by any SPI core or controller driver code.

Note that although all messages to a spi_device are handled in FIFO order, messages may go to different devices in other orders. Some device might be higher priority, or have various "hard" access time requirements, for example.

On detection of any fault during the transfer, processing of the entire message is aborted, and the device is deselected. Until returning from the associated message completion callback, no other spi_message queued to that device will be processed. (This rule applies equally to all the synchronous transfer calls, which are wrappers around this core asynchronous primitive.)

Return

zero on success, else a negative error code.

int **spi_sync**(struct spi_device * spi, struct spi_message * message)
    blocking/synchronous SPI data transfers

Parameters

struct spi_device * spi device with which data will be exchanged

struct spi_message * message describes the data transfers

Context

can sleep

Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout. Low-overhead controller drivers may DMA directly into and out of the message buffers.

Note that the SPI device's chip select is active during the message, and then is normally disabled between messages. Drivers for some frequently-used devices may want to minimize costs of selecting a chip, by leaving it selected in anticipation that the next message will go to the same chip. (That may increase power usage.)

Also, the caller is guaranteeing that the memory associated with the message will not be freed before this call returns.

Return

zero on success, else a negative error code.

int **spi_sync_locked**(struct spi_device * spi, struct spi_message * message)
    version of spi_sync with exclusive bus usage

Parameters

struct spi_device * spi device with which data will be exchanged

struct spi_message * message describes the data transfers

Context

can sleep

Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout. Low-overhead controller drivers may DMA directly into and out of the message buffers.

This call should be used by drivers that require exclusive access to the SPI bus. It has to be preceded by a spi_bus_lock call. The SPI bus must be released by a spi_bus_unlock call when the exclusive access is over.

Return

zero on success, else a negative error code.

int **spi_bus_lock**(struct spi_controller * ctlr)
obtain a lock for exclusive SPI bus usage

Parameters

struct spi_controller * ctlr SPI bus master that should be locked for exclusive bus access

Context

can sleep

Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout.

This call should be used by drivers that require exclusive access to the SPI bus. The SPI bus must be released by a spi_bus_unlock call when the exclusive access is over. Data transfer must be done by spi_sync_locked and spi_async_locked calls when the SPI bus lock is held.

Return

always zero.

int **spi_bus_unlock**(struct spi_controller * ctlr)
release the lock for exclusive SPI bus usage

Parameters

struct spi_controller * ctlr SPI bus master that was locked for exclusive bus access

Context

can sleep

Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout.

This call releases an SPI bus lock previously obtained by an spi_bus_lock call.

Return

always zero.

int **spi_write_then_read**(struct spi_device * spi, const void * txbuf, unsigned n_tx, void * rxbuf, unsigned n_rx)

SPI synchronous write followed by read

Parameters

struct spi_device * spi device with which data will be exchanged

const void * txbuf data to be written (need not be dma-safe)

unsigned n_tx size of txbuf, in bytes

void * rxbuf buffer into which data will be read (need not be dma-safe)

unsigned n_rx size of rxbuf, in bytes

Context

can sleep

Description

This performs a half duplex MicroWire style transaction with the device, sending txbuf and then reading rxbuf. The return value is zero for success, else a negative errno status code. This call may only be used from a context that may sleep.

Parameters to this routine are always copied using a small buffer. Performance-sensitive or bulk transfer code should instead use spi_{async,sync}() calls with dma-safe buffers.

Return

zero on success, else a negative error code.

CHAPTER

TWENTYFOUR

I²C AND SMBUS SUBSYSTEM

I²C (or without fancy typography, "I2C") is an acronym for the "Inter-IC" bus, a simple bus protocol which is widely used where low data rate communications suffice. Since it's also a licensed trademark, some vendors use another name (such as "Two-Wire Interface", TWI) for the same bus. I2C only needs two signals (SCL for clock, SDA for data), conserving board real estate and minimizing signal quality issues. Most I2C devices use seven bit addresses, and bus speeds of up to 400 kHz; there's a high speed extension (3.4 MHz) that's not yet found wide use. I2C is a multi-master bus; open drain signaling is used to arbitrate between masters, as well as to handshake and to synchronize clocks from slower clients.

The Linux I2C programming interfaces support the master side of bus interactions and the slave side. The programming interface is structured around two kinds of driver, and two kinds of device. An I2C "Adapter Driver" abstracts the controller hardware; it binds to a physical device (perhaps a PCI device or platform_device) and exposes a `struct i2c_adapter` representing each I2C bus segment it manages. On each I2C bus segment will be I2C devices represented by a `struct i2c_client`. Those devices will be bound to a `struct i2c_driver`, which should follow the standard Linux driver model. There are functions to perform various I2C protocol operations; at this writing all such functions are usable only from task context.

The System Management Bus (SMBus) is a sibling protocol. Most SMBus systems are also I2C conformant. The electrical constraints are tighter for SMBus, and it standardizes particular protocol messages and idioms. Controllers that support I2C can also support most SMBus operations, but SMBus controllers don't support all the protocol options that an I2C controller will. There are functions to perform various SMBus protocol operations, either using I2C primitives or by issuing SMBus commands to i2c_adapter devices which don't support those I2C operations.

int **i2c_master_recv**(const struct i2c_client * client, char * buf, int count)
issue a single I2C message in master receive mode

Parameters

const struct i2c_client * client Handle to slave device

char * buf Where to store data read from slave

int count How many bytes to read, must be less than 64k since msg.len is u16

Description

Returns negative errno, or else the number of bytes read.

int **i2c_master_recv_dmasafe**(const struct i2c_client * client, char * buf, int count)
issue a single I2C message in master receive mode using a DMA safe buffer

Parameters

const struct i2c_client * client Handle to slave device

char * buf Where to store data read from slave, must be safe to use with DMA

int count How many bytes to read, must be less than 64k since msg.len is u16

Description

Returns negative errno, or else the number of bytes read.

int **i2c_master_send**(const struct i2c_client * client, const char * buf, int count)
issue a single I2C message in master transmit mode

Parameters

const struct i2c_client * client Handle to slave device

const char * buf Data that will be written to the slave

int count How many bytes to write, must be less than 64k since msg.len is u16

Description

Returns negative errno, or else the number of bytes written.

int **i2c_master_send_dmasafe**(const struct i2c_client * client, const char * buf, int count)
issue a single I2C message in master transmit mode using a DMA safe buffer

Parameters

const struct i2c_client * client Handle to slave device

const char * buf Data that will be written to the slave, must be safe to use with DMA

int count How many bytes to write, must be less than 64k since msg.len is u16

Description

Returns negative errno, or else the number of bytes written.

struct **i2c_device_identity**
i2c client device identification

Definition

```
struct i2c_device_identity {
  u16 manufacturer_id;
#define I2C_DEVICE_ID_NXP_SEMICONDUCTORS                0;
#define I2C_DEVICE_ID_NXP_SEMICONDUCTORS_1              1;
#define I2C_DEVICE_ID_NXP_SEMICONDUCTORS_2              2;
#define I2C_DEVICE_ID_NXP_SEMICONDUCTORS_3              3;
#define I2C_DEVICE_ID_RAMTRON_INTERNATIONAL             4;
#define I2C_DEVICE_ID_ANALOG_DEVICES                    5;
#define I2C_DEVICE_ID_STMICROELECTRONICS                6;
```

(continues on next page)

(continued from previous page)

```
#define I2C_DEVICE_ID_ON_SEMICONDUCTOR                 7;
#define I2C_DEVICE_ID_SPRINTEK_CORPORATION             8;
#define I2C_DEVICE_ID_ESPROS_PHOTONICS_AG              9;
#define I2C_DEVICE_ID_FUJITSU_SEMICONDUCTOR           10;
#define I2C_DEVICE_ID_FLIR                            11;
#define I2C_DEVICE_ID_O2MICRO                         12;
#define I2C_DEVICE_ID_ATMEL                           13;
#define I2C_DEVICE_ID_NONE                          0xffff;
  u16 part_id;
  u8 die_revision;
};
```

Members

manufacturer_id 0 - 4095, database maintained by NXP

part_id 0 - 511, according to manufacturer

die_revision 0 - 7, according to manufacturer

struct **i2c_driver**
    represent an I2C device driver

Definition

```
struct i2c_driver {
  unsigned int class;
  int (*probe)(struct i2c_client *client, const struct i2c_device_id *id);
  int (*remove)(struct i2c_client *client);
  int (*probe_new)(struct i2c_client *client);
  void (*shutdown)(struct i2c_client *client);
  void (*alert)(struct i2c_client *client, enum i2c_alert_protocol␣
↪protocol, unsigned int data);
  int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);
  struct device_driver driver;
  const struct i2c_device_id *id_table;
  int (*detect)(struct i2c_client *client, struct i2c_board_info *info);
  const unsigned short *address_list;
  struct list_head clients;
  bool disable_i2c_core_irq_mapping;
};
```

Members

class What kind of i2c device we instantiate (for detect)

probe Callback for device binding - soon to be deprecated

remove Callback for device unbinding

probe_new New callback for device binding

shutdown Callback for device shutdown

alert Alert callback, for example for the SMBus alert protocol

command Callback for bus-wide signaling (optional)

driver Device driver model driver

id_table List of I2C devices supported by this driver

detect Callback for device detection

address_list The I2C addresses to probe (for detect)

clients List of detected clients we created (for i2c-core use only)

disable_i2c_core_irq_mapping Tell the i2c-core to not do irq-mapping

Description

The driver.owner field should be set to the module owner of this driver. The driver.name field should be set to the name of this driver.

For automatic device detection, both **detect** and **address_list** must be defined. **class** should also be set, otherwise only devices forced with module parameters will be created. The detect function must fill at least the name field of the i2c_board_info structure it is handed upon successful detection, and possibly also the flags field.

If **detect** is missing, the driver will still work fine for enumerated devices. Detected devices simply won't be supported. This is expected for the many I2C/SMBus devices which can't be detected reliably, and the ones which can always be enumerated in practice.

The i2c_client structure which is handed to the **detect** callback is not a real i2c_client. It is initialized just enough so that you can call i2c_smbus_read_byte_data and friends on it. Don't do anything else with it. In particular, calling dev_dbg and friends on it is not allowed.

struct **i2c_client**
represent an I2C slave device

Definition

```
struct i2c_client {
  unsigned short flags;
#define I2C_CLIENT_PEC          0x04    ;
#define I2C_CLIENT_TEN          0x10    ;
#define I2C_CLIENT_SLAVE        0x20    ;
#define I2C_CLIENT_HOST_NOTIFY  0x40    ;
#define I2C_CLIENT_WAKE         0x80    ;
#define I2C_CLIENT_SCCB         0x9000  ;
  unsigned short addr;
  char name[I2C_NAME_SIZE];
  struct i2c_adapter *adapter;
  struct device dev;
  int init_irq;
  int irq;
  struct list_head detected;
#if IS_ENABLED(CONFIG_I2C_SLAVE);
  i2c_slave_cb_t slave_cb;
#endif;
};
```

Members

flags see I2C_CLIENT_* for possible flags

addr Address used on the I2C bus connected to the parent adapter.

name Indicates the type of the device, usually a chip name that's generic enough to hide second-sourcing and compatible revisions.

adapter manages the bus segment hosting this I2C device

dev Driver model device node for the slave.

init_irq IRQ that was set at initialization

irq indicates the IRQ generated by this device (if any)

detected member of an i2c_driver.clients list or i2c-core's userspace_devices list

slave_cb Callback when I2C slave mode of an adapter is used. The adapter calls it to pass on slave events to the slave driver.

Description

An i2c_client identifies a single device (i.e. chip) connected to an i2c bus. The behaviour exposed to Linux is defined by the driver managing the device.

struct **i2c_board_info**
template for device creation

Definition

```
struct i2c_board_info {
  char type[I2C_NAME_SIZE];
  unsigned short  flags;
  unsigned short  addr;
  const char      *dev_name;
  void *platform_data;
  struct device_node *of_node;
  struct fwnode_handle *fwnode;
  const struct property_entry *properties;
  const struct resource *resources;
  unsigned int    num_resources;
  int irq;
};
```

Members

type chip type, to initialize i2c_client.name

flags to initialize i2c_client.flags

addr stored in i2c_client.addr

dev_name Overrides the default <busnr>-<addr> dev_name if set

platform_data stored in i2c_client.dev.platform_data

of_node pointer to OpenFirmware device node

fwnode device node supplied by the platform firmware

properties additional device properties for the device

resources resources associated with the device

num_resources number of resources in the **resources** array

irq stored in i2c_client.irq

Description

I2C doesn't actually support hardware probing, although controllers and devices may be able to use I2C_SMBUS_QUICK to tell whether or not there's a device at a given address. Drivers commonly need more information than that, such as chip type, configuration, associated IRQ, and so on.

i2c_board_info is used to build tables of information listing I2C devices that are present. This information is used to grow the driver model tree. For mainboards this is done statically using `i2c_register_board_info()`; bus numbers identify adapters that aren't yet available. For add-on boards, `i2c_new_client_device()` does this dynamically with the adapter already known.

I2C_BOARD_INFO(dev_type, dev_addr)
macro used to list an i2c device and its address

Parameters

dev_type identifies the device type

dev_addr the device's address on the bus.

Description

This macro initializes essential fields of a struct i2c_board_info, declaring what has been provided on a particular board. Optional fields (such as associated irq, or device-specific platform_data) are provided using conventional syntax.

struct **i2c_algorithm**
represent I2C transfer method

Definition

```
struct i2c_algorithm {
  int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs, int␣
↪num);
  int (*master_xfer_atomic)(struct i2c_adapter *adap, struct i2c_msg *msgs,
↪ int num);
  int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,unsigned short␣
↪flags, char read_write, u8 command, int size, union i2c_smbus_data␣
↪*data);
  int (*smbus_xfer_atomic)(struct i2c_adapter *adap, u16 addr,unsigned␣
↪short flags, char read_write, u8 command, int size, union i2c_smbus_data␣
↪*data);
  u32 (*functionality)(struct i2c_adapter *adap);
#if IS_ENABLED(CONFIG_I2C_SLAVE);
  int (*reg_slave)(struct i2c_client *client);
  int (*unreg_slave)(struct i2c_client *client);
#endif;
};
```

Members

master_xfer Issue a set of i2c transactions to the given I2C adapter defined by the msgs array, with num messages available to transfer via the adapter specified by adap.

master_xfer_atomic same as **master_xfer**. Yet, only using atomic context so e.g. PMICs can be accessed very late before shutdown. Optional.

smbus_xfer Issue smbus transactions to the given I2C adapter. If this is not present, then the bus layer will try and convert the SMBus calls into I2C transfers instead.

smbus_xfer_atomic same as **smbus_xfer**. Yet, only using atomic context so e.g. PMICs can be accessed very late before shutdown. Optional.

functionality Return the flags that this algorithm/adapter pair supports from the I2C_FUNC_* flags.

reg_slave Register given client to I2C slave mode of this adapter

unreg_slave Unregister given client from I2C slave mode of this adapter

Description

The following structs are for those who like to implement new bus drivers: i2c_algorithm is the interface to a class of hardware solutions which can be addressed using the same bus algorithms - i.e. bit-banging or the PCF8584 to name two of the most common.

The return codes from the master_xfer{_atomic} fields should indicate the type of error code that occurred during the transfer, as documented in the Kernel Documentation file Documentation/i2c/fault-codes.rst.

struct **i2c_lock_operations**
represent I2C locking operations

Definition

```
struct i2c_lock_operations {
  void (*lock_bus)(struct i2c_adapter *adapter, unsigned int flags);
  int (*trylock_bus)(struct i2c_adapter *adapter, unsigned int flags);
  void (*unlock_bus)(struct i2c_adapter *adapter, unsigned int flags);
};
```

Members

lock_bus Get exclusive access to an I2C bus segment

trylock_bus Try to get exclusive access to an I2C bus segment

unlock_bus Release exclusive access to an I2C bus segment

Description

The main operations are wrapped by i2c_lock_bus and i2c_unlock_bus.

struct **i2c_timings**
I2C timing information

Definition

```
struct i2c_timings {
  u32 bus_freq_hz;
  u32 scl_rise_ns;
  u32 scl_fall_ns;
```

(continues on next page)

(continued from previous page)

```
  u32 scl_int_delay_ns;
  u32 sda_fall_ns;
  u32 sda_hold_ns;
  u32 digital_filter_width_ns;
  u32 analog_filter_cutoff_freq_hz;
};
```

Members

bus_freq_hz the bus frequency in Hz

scl_rise_ns time SCL signal takes to rise in ns; t(r) in the I2C specification

scl_fall_ns time SCL signal takes to fall in ns; t(f) in the I2C specification

scl_int_delay_ns time IP core additionally needs to setup SCL in ns

sda_fall_ns time SDA signal takes to fall in ns; t(f) in the I2C specification

sda_hold_ns time IP core additionally needs to hold SDA in ns

digital_filter_width_ns width in ns of spikes on i2c lines that the IP core digital filter can filter out

analog_filter_cutoff_freq_hz threshold frequency for the low pass IP core analog filter

struct **i2c_bus_recovery_info**
I2C bus recovery information

Definition

```
struct i2c_bus_recovery_info {
  int (*recover_bus)(struct i2c_adapter *adap);
  int (*get_scl)(struct i2c_adapter *adap);
  void (*set_scl)(struct i2c_adapter *adap, int val);
  int (*get_sda)(struct i2c_adapter *adap);
  void (*set_sda)(struct i2c_adapter *adap, int val);
  int (*get_bus_free)(struct i2c_adapter *adap);
  void (*prepare_recovery)(struct i2c_adapter *adap);
  void (*unprepare_recovery)(struct i2c_adapter *adap);
  struct gpio_desc *scl_gpiod;
  struct gpio_desc *sda_gpiod;
};
```

Members

recover_bus Recover routine. Either pass driver's recover_bus() routine, or i2c_generic_scl_recovery().

get_scl This gets current value of SCL line. Mandatory for generic SCL recovery. Populated internally for generic GPIO recovery.

set_scl This sets/clears the SCL line. Mandatory for generic SCL recovery. Populated internally for generic GPIO recovery.

get_sda This gets current value of SDA line. This or set_sda() is mandatory for generic SCL recovery. Populated internally, if sda_gpio is a valid GPIO, for generic GPIO recovery.

set_sda This sets/clears the SDA line. This or get_sda() is mandatory for generic SCL recovery. Populated internally, if sda_gpio is a valid GPIO, for generic GPIO recovery.

get_bus_free Returns the bus free state as seen from the IP core in case it has a more complex internal logic than just reading SDA. Optional.

prepare_recovery This will be called before starting recovery. Platform may configure padmux here for SDA/SCL line or something else they want.

unprepare_recovery This will be called after completing recovery. Platform may configure padmux here for SDA/SCL line or something else they want.

scl_gpiod gpiod of the SCL line. Only required for GPIO recovery.

sda_gpiod gpiod of the SDA line. Only required for GPIO recovery.

struct **i2c_adapter_quirks**
describe flaws of an i2c adapter

Definition

```
struct i2c_adapter_quirks {
  u64 flags;
  int max_num_msgs;
  u16 max_write_len;
  u16 max_read_len;
  u16 max_comb_1st_msg_len;
  u16 max_comb_2nd_msg_len;
};
```

Members

flags see I2C_AQ_* for possible flags and read below

max_num_msgs maximum number of messages per transfer

max_write_len maximum length of a write message

max_read_len maximum length of a read message

max_comb_1st_msg_len maximum length of the first msg in a combined message

max_comb_2nd_msg_len maximum length of the second msg in a combined message

Description

Note about combined messages: Some I2C controllers can only send one message per transfer, plus something called combined message or write-then-read. This is (usually) a small write message followed by a read message and barely enough to access register based devices like EEPROMs. There is a flag to support this mode. It implies max_num_msg = 2 and does the length checks with max_comb_*_len because combined message mode usually has its own limitations. Because of HW implementations, some controllers can actually do write-then-anything or other variants. To support that, write-then-read has been broken out into smaller bits like write-first and read-second which can be combined as needed.

void **i2c_lock_bus**(struct i2c_adapter * adapter, unsigned int flags)
Get exclusive access to an I2C bus segment

Parameters

struct i2c_adapter * adapter Target I2C bus segment

unsigned int flags I2C_LOCK_ROOT_ADAPTER locks the root i2c adapter, I2C_LOCK_SEGMENT locks only this branch in the adapter tree

int **i2c_trylock_bus**(struct i2c_adapter * adapter, unsigned int flags)
Try to get exclusive access to an I2C bus segment

Parameters

struct i2c_adapter * adapter Target I2C bus segment

unsigned int flags I2C_LOCK_ROOT_ADAPTER tries to locks the root i2c adapter, I2C_LOCK_SEGMENT tries to lock only this branch in the adapter tree

Return

true if the I2C bus segment is locked, false otherwise

void **i2c_unlock_bus**(struct i2c_adapter * adapter, unsigned int flags)
Release exclusive access to an I2C bus segment

Parameters

struct i2c_adapter * adapter Target I2C bus segment

unsigned int flags I2C_LOCK_ROOT_ADAPTER unlocks the root i2c adapter, I2C_LOCK_SEGMENT unlocks only this branch in the adapter tree

void **i2c_mark_adapter_suspended**(struct i2c_adapter * adap)
Report suspended state of the adapter to the core

Parameters

struct i2c_adapter * adap Adapter to mark as suspended

Description

When using this helper to mark an adapter as suspended, the core will reject further transfers to this adapter. The usage of this helper is optional but recommended for devices having distinct handlers for system suspend and runtime suspend. More complex devices are free to implement custom solutions to reject transfers when suspended.

void **i2c_mark_adapter_resumed**(struct i2c_adapter * adap)
Report resumed state of the adapter to the core

Parameters

struct i2c_adapter * adap Adapter to mark as resumed

Description

When using this helper to mark an adapter as resumed, the core will allow further transfers to this adapter. See also further notes to **i2c_mark_adapter_suspended()**.

bool **i2c_check_quirks**(struct i2c_adapter * adap, u64 quirks)
Function for checking the quirk flags in an i2c adapter

Parameters

struct i2c_adapter * adap i2c adapter

u64 quirks quirk flags

Return

true if the adapter has all the specified quirk flags, false if not

module_i2c_driver(__i2c_driver)
Helper macro for registering a modular I2C driver

Parameters

__i2c_driver i2c_driver struct

Description

Helper macro for I2C drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init()` and `module_exit()`

builtin_i2c_driver(__i2c_driver)
Helper macro for registering a builtin I2C driver

Parameters

__i2c_driver i2c_driver struct

Description

Helper macro for I2C drivers which do not do anything special in their init. This eliminates a lot of boilerplate. Each driver may only use this macro once, and calling it replaces device_initcall().

int **i2c_register_board_info**(int busnum, struct i2c_board_info const * info, unsigned len)
statically declare I2C devices

Parameters

int busnum identifies the bus to which these devices belong

struct i2c_board_info const * info vector of i2c device descriptors

unsigned len how many descriptors in the vector; may be zero to reserve the specified bus number.

Description

Systems using the Linux I2C driver stack can declare tables of board info while they initialize. This should be done in board-specific init code near arch_initcall() time, or equivalent, before any I2C adapter driver is registered. For example, mainboard init code could define several devices, as could the init code for each daughtercard in a board stack.

The I2C devices will be created later, after the adapter for the relevant bus has been registered. After that moment, standard driver model tools are used to bind “new style” I2C drivers to the devices. The bus number for any device declared using this routine is not available for dynamic allocation.

The board info passed can safely be __initdata, but be careful of embedded pointers (for platform_data, functions, etc) since that won't be copied. Device properties are deep-copied though.

struct i2c_client * **i2c_verify_client**(struct device * dev)
return parameter as i2c_client, or NULL

Parameters

struct device * dev device, probably from some driver model iterator

Description

When traversing the driver model tree, perhaps using driver model iterators like **device_for_each_child()**, you can't assume very much about the nodes you find. Use this function to avoid oopses caused by wrongly treating some non-I2C device as an i2c_client.

struct i2c_client * **i2c_new_client_device**(struct i2c_adapter * adap, struct i2c_board_info const * info)
instantiate an i2c device

Parameters

struct i2c_adapter * adap the adapter managing the device

struct i2c_board_info const * info describes one I2C device; bus_num is ignored

Context

can sleep

Description

Create an i2c device. Binding is handled through driver model probe()/remove() methods. A driver may be bound to this device when we return from this function, or any later moment (e.g. maybe hotplugging will load the driver module). This call is not appropriate for use by mainboard initialization logic, which usually runs during an arch_initcall() long before any i2c_adapter could exist.

This returns the new i2c client, which may be saved for later use with `i2c_unregister_device()`; or an ERR_PTR to describe the error.

void **i2c_unregister_device**(struct i2c_client * client)
reverse effect of i2c_new_*_device()

Parameters

struct i2c_client * client value returned from i2c_new_*_device()

Context

can sleep

struct i2c_client * **i2c_new_dummy_device**(struct i2c_adapter * adapter, u16 address)
return a new i2c device bound to a dummy driver

Parameters

struct i2c_adapter * adapter the adapter managing the device

u16 address seven bit address to be used

Context

can sleep

Description

This returns an I2C client bound to the “dummy” driver, intended for use with devices that consume multiple addresses. Examples of such chips include various EEPROMS (like 24c04 and 24c08 models).

These dummy devices have two main uses. First, most I2C and SMBus calls except `i2c_transfer()` need a client handle; the dummy will be that handle. And second, this prevents the specified address from being bound to a different driver.

This returns the new i2c client, which should be saved for later use with `i2c_unregister_device()`; or an ERR_PTR to describe the error.

struct i2c_client * **devm_i2c_new_dummy_device**(struct device * dev, struct i2c_adapter * adapter, u16 address)

return a new i2c device bound to a dummy driver

Parameters

struct device * dev device the managed resource is bound to

struct i2c_adapter * adapter the adapter managing the device

u16 address seven bit address to be used

Context

can sleep

Description

This is the device-managed version of **i2c_new_dummy_device**. It returns the new i2c client or an ERR_PTR in case of an error.

struct i2c_client * **i2c_new_ancillary_device**(struct i2c_client * client, const char * name, u16 default_addr)

Helper to get the instantiated secondary address and create the associated device

Parameters

struct i2c_client * client Handle to the primary client

const char * name Handle to specify which secondary address to get

u16 default_addr Used as a fallback if no secondary address was specified

Context

can sleep

Description

I2C clients can be composed of multiple I2C slaves bound together in a single component. The I2C client driver then binds to the master I2C slave and needs to create I2C dummy clients to communicate with all the other slaves.

This function creates and returns an I2C dummy client whose I2C address is retrieved from the platform firmware based on the given slave name. If no address is specified by the firmware default_addr is used.

On DT-based platforms the address is retrieved from the "reg" property entry cell whose "reg-names" value matches the slave name.

This returns the new i2c client, which should be saved for later use with `i2c_unregister_device()`; or an ERR_PTR to describe the error.

struct i2c_adapter * **i2c_verify_adapter**(struct device * dev)
return parameter as i2c_adapter or NULL

Parameters

struct device * dev device, probably from some driver model iterator

Description

When traversing the driver model tree, perhaps using driver model iterators like **device_for_each_child()**, you can't assume very much about the nodes you find. Use this function to avoid oopses caused by wrongly treating some non-I2C device as an i2c_adapter.

int **i2c_handle_smbus_host_notify**(struct i2c_adapter * adap, unsigned short addr)
Forward a Host Notify event to the correct I2C client.

Parameters

struct i2c_adapter * adap the adapter

unsigned short addr the I2C address of the notifying device

Context

can't sleep

Description

Helper function to be called from an I2C bus driver's interrupt handler. It will schedule the Host Notify IRQ.

int **i2c_add_adapter**(struct i2c_adapter * adapter)
declare i2c adapter, use dynamic bus number

Parameters

struct i2c_adapter * adapter the adapter to add

Context

can sleep

Description

This routine is used to declare an I2C adapter when its bus number doesn't matter or when its bus number is specified by an dt alias. Examples of bases when the bus number doesn't matter: I2C adapters dynamically added by USB links or PCI plugin cards.

When this returns zero, a new bus number was allocated and stored in adap->nr, and the specified adapter became available for clients. Otherwise, a negative errno value is returned.

int **i2c_add_numbered_adapter**(struct i2c_adapter * adap)
declare i2c adapter, use static bus number

Parameters

struct i2c_adapter * adap the adapter to register (with adap->nr initialized)

Context

can sleep

Description

This routine is used to declare an I2C adapter when its bus number matters. For example, use it for I2C adapters from system-on-chip CPUs, or otherwise built in to the system's mainboard, and where i2c_board_info is used to properly configure I2C devices.

If the requested bus number is set to -1, then this function will behave identically to i2c_add_adapter, and will dynamically assign a bus number.

If no devices have pre-been declared for this bus, then be sure to register the adapter before any dynamically allocated ones. Otherwise the required bus ID may not be available.

When this returns zero, the specified adapter became available for clients using the bus number provided in adap->nr. Also, the table of I2C devices pre-declared using `i2c_register_board_info()` is scanned, and the appropriate driver model device nodes are created. Otherwise, a negative errno value is returned.

void **i2c_del_adapter**(struct i2c_adapter * adap)
unregister I2C adapter

Parameters

struct i2c_adapter * adap the adapter being unregistered

Context

can sleep

Description

This unregisters an I2C adapter which was previously registered by **i2c_add_adapter** or **i2c_add_numbered_adapter**.

void **i2c_parse_fw_timings**(struct device * dev, struct i2c_timings * t,
bool use_defaults)
get I2C related timing parameters from firmware

Parameters

struct device * dev The device to scan for I2C timing properties

struct i2c_timings * t the i2c_timings struct to be filled with values

bool use_defaults bool to use sane defaults derived from the I2C specification when properties are not found, otherwise don't update

Description

Scan the device for the generic I2C properties describing timing parameters for the signal and fill the given struct with the results. If a property was not found and use_defaults was true, then maximum timings are assumed which are derived from the I2C specification. If use_defaults is not used, the results will be as before, so drivers can apply their own defaults before calling this helper. The latter is mainly intended for avoiding regressions of existing drivers which want to switch to this function. New drivers almost always should use the defaults.

void **i2c_del_driver**(struct i2c_driver * driver)
unregister I2C driver

Parameters

struct i2c_driver * driver the driver being unregistered

Context

can sleep

int **__i2c_transfer**(struct i2c_adapter * adap, struct i2c_msg * msgs, int num)
unlocked flavor of i2c_transfer

Parameters

struct i2c_adapter * adap Handle to I2C bus

struct i2c_msg * msgs One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

int num Number of messages to be executed.

Description

Returns negative errno, else the number of messages executed.

Adapter lock must be held when calling this function. No debug logging takes place. adap->algo->master_xfer existence isn't checked.

int **i2c_transfer**(struct i2c_adapter * adap, struct i2c_msg * msgs, int num)
execute a single or combined I2C message

Parameters

struct i2c_adapter * adap Handle to I2C bus

struct i2c_msg * msgs One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

int num Number of messages to be executed.

Description

Returns negative errno, else the number of messages executed.

Note that there is no requirement that each message be sent to the same slave address, although that is the most common model.

int **i2c_transfer_buffer_flags**(const struct i2c_client * client, char * buf, int count, u16 flags)
issue a single I2C message transferring data to/from a buffer

Parameters

const struct i2c_client * client Handle to slave device

char * buf Where the data is stored

int count How many bytes to transfer, must be less than 64k since msg.len is u16

u16 flags The flags to be used for the message, e.g. I2C_M_RD for reads

Description

Returns negative errno, or else the number of bytes transferred.

int **i2c_get_device_id**(const struct i2c_client * client, struct i2c_device_identity * id)
get manufacturer, part id and die revision of a device

Parameters

const struct i2c_client * client The device to query

struct i2c_device_identity * id The queried information

Description

Returns negative errno on error, zero on success.

u8 * **i2c_get_dma_safe_msg_buf**(struct i2c_msg * msg, unsigned int threshold)
get a DMA safe buffer for the given i2c_msg

Parameters

struct i2c_msg * msg the message to be checked

unsigned int threshold the minimum number of bytes for which using DMA makes sense. Should at least be 1.

Return

NULL if a DMA safe buffer was not obtained. Use msg->buf with PIO.
Or a valid pointer to be used with DMA. After use, release it by calling `i2c_put_dma_safe_msg_buf()`.

Description

This function must only be called from process context!

void **i2c_put_dma_safe_msg_buf**(u8 * buf, struct i2c_msg * msg, bool xferred)
release DMA safe buffer and sync with i2c_msg

Parameters

u8 * buf the buffer obtained from `i2c_get_dma_safe_msg_buf()`. May be NULL.

struct i2c_msg * msg the message which the buffer corresponds to

bool xferred bool saying if the message was transferred

s32 **i2c_smbus_read_byte**(const struct i2c_client * client)
SMBus “receive byte” protocol

Parameters

const struct i2c_client * client Handle to slave device

Description

This executes the SMBus “receive byte” protocol, returning negative errno else the byte received from the device.

s32 **i2c_smbus_write_byte**(const struct i2c_client * client, u8 value)
SMBus “send byte” protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 value Byte to be sent

Description

This executes the SMBus “send byte” protocol, returning negative errno else zero on success.

s32 **i2c_smbus_read_byte_data**(const struct i2c_client * client, u8 command)
SMBus “read byte” protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

Description

This executes the SMBus “read byte” protocol, returning negative errno else a data byte received from the device.

s32 **i2c_smbus_write_byte_data**(const struct i2c_client * client, u8 command, u8 value)
SMBus “write byte” protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u8 value Byte being written

Description

This executes the SMBus “write byte” protocol, returning negative errno else zero on success.

s32 **i2c_smbus_read_word_data**(const struct i2c_client * client, u8 command)
SMBus “read word” protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

Description

This executes the SMBus “read word” protocol, returning negative errno else a 16-bit unsigned “word” received from the device.

s32 **i2c_smbus_write_word_data**(const struct i2c_client * client, u8 command, u16 value)
SMBus “write word” protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u16 value 16-bit “word” being written

Description

This executes the SMBus “write word” protocol, returning negative errno else zero on success.

s32 **i2c_smbus_read_block_data**(const struct i2c_client * client, u8 command, u8 * values)
SMBus “block read” protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u8 * values Byte array into which data will be read; big enough to hold the data returned by the slave. SMBus allows at most 32 bytes.

Description

This executes the SMBus “block read” protocol, returning negative errno else the number of data bytes in the slave’s response.

Note that using this function requires that the client’s adapter support the I2C_FUNC_SMBUS_READ_BLOCK_DATA functionality. Not all adapter drivers support this; its emulation through I2C messaging relies on a specific mechanism (I2C_M_RECV_LEN) which may not be implemented.

s32 **i2c_smbus_write_block_data**(const struct i2c_client * client, u8 command, u8 length, const u8 * values)
SMBus “block write” protocol

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u8 length Size of data block; SMBus allows at most 32 bytes

const u8 * values Byte array which will be written.

Description

This executes the SMBus “block write” protocol, returning negative errno else zero on success.

s32 **i2c_smbus_xfer**(struct i2c_adapter * adapter, u16 addr, unsigned short flags, char read_write, u8 command, int protocol, union i2c_smbus_data * data)
execute SMBus protocol operations

Parameters

struct i2c_adapter * adapter Handle to I2C bus

u16 addr Address of SMBus slave on that bus

unsigned short flags I2C_CLIENT_* flags (usually zero or I2C_CLIENT_PEC)

char read_write I2C_SMBUS_READ or I2C_SMBUS_WRITE

u8 command Byte interpreted by slave, for protocols which use such bytes

int protocol SMBus protocol operation to execute, such as I2C_SMBUS_PROC_CALL

union i2c_smbus_data * data Data to be read or written

Description

This executes an SMBus protocol operation, and returns a negative errno code else zero on success.

s32 **i2c_smbus_read_i2c_block_data_or_emulated**(const struct i2c_client * client, u8 command, u8 length, u8 * values)
read block or emulate

Parameters

const struct i2c_client * client Handle to slave device

u8 command Byte interpreted by slave

u8 length Size of data block; SMBus allows at most I2C_SMBUS_BLOCK_MAX bytes

u8 * values Byte array into which data will be read; big enough to hold the data returned by the slave. SMBus allows at most I2C_SMBUS_BLOCK_MAX bytes.

Description

This executes the SMBus “block read” protocol if supported by the adapter. If block read is not supported, it emulates it using either word or byte read protocols depending on availability.

The addresses of the I2C slave device that are accessed with this function must be mapped to a linear region, so that a block read will have the same effect as a byte read. Before using this function you must double-check if the I2C slave does support exchanging a block transfer with a byte transfer.

struct i2c_client * **i2c_new_smbus_alert_device**(struct i2c_adapter * adapter, struct i2c_smbus_alert_setup * setup)
get ara client for SMBus alert support

Parameters

struct i2c_adapter * adapter the target adapter

struct i2c_smbus_alert_setup * setup setup data for the SMBus alert handler

Context

can sleep

Description

Setup handling of the SMBus alert protocol on a given I2C bus segment.

Handling can be done either through our IRQ handler, or by the adapter (from its handler, periodic polling, or whatever).

This returns the ara client, which should be saved for later use with i2c_handle_smbus_alert() and ultimately `i2c_unregister_device()`; or an ERRPTR to indicate an error.

CHAPTER

TWENTYFIVE

IPMB DRIVER FOR A SATELLITE MC

The Intelligent Platform Management Bus or IPMB, is an I2C bus that provides a standardized interconnection between different boards within a chassis. This interconnection is between the baseboard management (BMC) and chassis electronics. IPMB is also associated with the messaging protocol through the IPMB bus.

The devices using the IPMB are usually management controllers that perform management functions such as servicing the front panel interface, monitoring the baseboard, hot-swapping disk drivers in the system chassis, etc⋯

When an IPMB is implemented in the system, the BMC serves as a controller to give system software access to the IPMB. The BMC sends IPMI requests to a device (usually a Satellite Management Controller or Satellite MC) via IPMB and the device sends a response back to the BMC.

For more information on IPMB and the format of an IPMB message, refer to the IPMB and IPMI specifications.

25.1 IPMB driver for Satellite MC

ipmb-dev-int - This is the driver needed on a Satellite MC to receive IPMB messages from a BMC and send a response back. This driver works with the I2C driver and a userspace program such as OpenIPMI:

1) It is an I2C slave backend driver. So, it defines a callback function to set the Satellite MC as an I2C slave. This callback function handles the received IPMI requests.

2) It defines the read and write functions to enable a user space program (such as OpenIPMI) to communicate with the kernel.

25.2 Load the IPMB driver

The driver needs to be loaded at boot time or manually first. First, make sure you have the following in your config file: CONFIG_IPMB_DEVICE_INTERFACE=y

1) If you want the driver to be loaded at boot time:

a) Add this entry to your ACPI table, under the appropriate SMBus:

```
Device (SMB0) // Example SMBus host controller
{
Name (_HID, "<Vendor-Specific HID>") // Vendor-Specific HID
Name (_UID, 0) // Unique ID of particular host controller
:
:
  Device (IPMB)
  {
    Name (_HID, "IPMB0001") // IPMB device interface
    Name (_UID, 0) // Unique device identifier
  }
}
```

b) Example for device tree:

```
&i2c2 {
       status = "okay";

       ipmb@10 {
               compatible = "ipmb-dev";
               reg = <0x10>;
               i2c-protocol;
       };
};
```

If xmit of data to be done using raw i2c block vs smbus then “i2c-protocol” needs to be defined as above.

2) Manually from Linux:

```
modprobe ipmb-dev-int
```

25.3 Instantiate the device

After loading the driver, you can instantiate the device as described in ‘Documentation/i2c/instantiating-devices.rst’. If you have multiple BMCs, each connected to your Satellite MC via a different I2C bus, you can instantiate a device for each of those BMCs.

The name of the instantiated device contains the I2C bus number associated with it as follows:

```
BMC1 ------ IPMB/I2C bus 1 ---------|   /dev/ipmb-1
                                Satellite MC
BMC1 ------ IPMB/I2C bus 2 ---------|   /dev/ipmb-2
```

For instance, you can instantiate the ipmb-dev-int device from user space at the 7 bit address 0x10 on bus 2:

```
# echo ipmb-dev 0x1010 > /sys/bus/i2c/devices/i2c-2/new_device
```

This will create the device file /dev/ipmb-2, which can be accessed by the user space program. The device needs to be instantiated before running the user space program.

CHAPTER

TWENTYSIX

THE LINUX IPMI DRIVER

Author Corey Minyard <minyard@mvista.com> / <minyard@acm.org>

The Intelligent Platform Management Interface, or IPMI, is a standard for controlling intelligent devices that monitor a system. It provides for dynamic discovery of sensors in the system and the ability to monitor the sensors and be informed when the sensor's values change or go outside certain boundaries. It also has a standardized database for field-replaceable units (FRUs) and a watchdog timer.

To use this, you need an interface to an IPMI controller in your system (called a Baseboard Management Controller, or BMC) and management software that can use the IPMI system.

This document describes how to use the IPMI driver for Linux. If you are not familiar with IPMI itself, see the web site at http://www.intel.com/design/servers/ipmi/index.htm. IPMI is a big subject and I can't cover it all here!

26.1 Configuration

The Linux IPMI driver is modular, which means you have to pick several things to have it work right depending on your hardware. Most of these are available in the 'Character Devices' menu then the IPMI menu.

No matter what, you must pick 'IPMI top-level message handler' to use IPMI. What you do beyond that depends on your needs and hardware.

The message handler does not provide any user-level interfaces. Kernel code (like the watchdog) can still use it. If you need access from userland, you need to select 'Device interface for IPMI' if you want access through a device driver.

The driver interface depends on your hardware. If your system properly provides the SMBIOS info for IPMI, the driver will detect it and just work. If you have a board with a standard interface (These will generally be either "KCS", "SMIC", or "BT", consult your hardware manual), choose the 'IPMI SI handler' option. A driver also exists for direct I2C access to the IPMI management controller. Some boards support this, but it is unknown if it will work on every board. For this, choose 'IPMI SMBus handler', but be ready to try to do some figuring to see if it will work on your system if the SMBIOS/APCI information is wrong or not present. It is fairly safe to have both these enabled and let the drivers auto-detect what is present.

You should generally enable ACPI on your system, as systems with IPMI can have ACPI tables describing them.

If you have a standard interface and the board manufacturer has done their job correctly, the IPMI controller should be automatically detected (via ACPI or SMBIOS tables) and should just work. Sadly, many boards do not have this information. The driver attempts standard defaults, but they may not work. If you fall into this situation, you need to read the section below named 'The SI Driver' or "The SMBus Driver" on how to hand-configure your system.

IPMI defines a standard watchdog timer. You can enable this with the 'IPMI Watchdog Timer' config option. If you compile the driver into the kernel, then via a kernel command-line option you can have the watchdog timer start as soon as it initializes. It also have a lot of other options, see the 'Watchdog' section below for more details. Note that you can also have the watchdog continue to run if it is closed (by default it is disabled on close). Go into the 'Watchdog Cards' menu, enable 'Watchdog Timer Support', and enable the option 'Disable watchdog shutdown on close'.

IPMI systems can often be powered off using IPMI commands. Select 'IPMI Poweroff' to do this. The driver will auto-detect if the system can be powered off by IPMI. It is safe to enable this even if your system doesn't support this option. This works on ATCA systems, the Radisys CPI1 card, and any IPMI system that supports standard chassis management commands.

If you want the driver to put an event into the event log on a panic, enable the 'Generate a panic event to all BMCs on a panic' option. If you want the whole panic string put into the event log using OEM events, enable the 'Generate OEM events containing the panic string' option. You can also enable these dynamically by setting the module parameter named "panic_op"in the ipmi_msghandler module to "event" or "string". Setting that parameter to "none" disables this function.

26.2 Basic Design

The Linux IPMI driver is designed to be very modular and flexible, you only need to take the pieces you need and you can use it in many different ways. Because of that, it's broken into many chunks of code. These chunks (by module name) are:

ipmi_msghandler - This is the central piece of software for the IPMI system. It handles all messages, message timing, and responses. The IPMI users tie into this, and the IPMI physical interfaces (called System Management Interfaces, or SMIs) also tie in here. This provides the kernelland interface for IPMI, but does not provide an interface for use by application processes.

ipmi_devintf - This provides a userland IOCTL interface for the IPMI driver, each open file for this device ties in to the message handler as an IPMI user.

ipmi_si - A driver for various system interfaces. This supports KCS, SMIC, and BT interfaces. Unless you have an SMBus interface or your own custom interface, you probably need to use this.

ipmi_ssif - A driver for accessing BMCs on the SMBus. It uses the I2C kernel driver's SMBus interfaces to send and receive IPMI messages over the SMBus.

ipmi_powernv - A driver for access BMCs on POWERNV systems.

ipmi_watchdog - IPMI requires systems to have a very capable watchdog timer. This driver implements the standard Linux watchdog timer interface on top of the IPMI message handler.

ipmi_poweroff - Some systems support the ability to be turned off via IPMI commands.

bt-bmc - This is not part of the main driver, but instead a driver for accessing a BMC-side interface of a BT interface. It is used on BMCs running Linux to provide an interface to the host.

These are all individually selectable via configuration options.

Much documentation for the interface is in the include files. The IPMI include files are:

linux/ipmi.h - Contains the user interface and IOCTL interface for IPMI.

linux/ipmi_smi.h - Contains the interface for system management interfaces (things that interface to IPMI controllers) to use.

linux/ipmi_msgdefs.h - General definitions for base IPMI messaging.

26.3 Addressing

The IPMI addressing works much like IP addresses, you have an overlay to handle the different address types. The overlay is:

```
struct ipmi_addr
{
      int   addr_type;
      short channel;
      char  data[IPMI_MAX_ADDR_SIZE];
};
```

The addr_type determines what the address really is. The driver currently understands two different types of addresses.

"System Interface" addresses are defined as:

```
struct ipmi_system_interface_addr
{
      int   addr_type;
      short channel;
};
```

and the type is IPMI_SYSTEM_INTERFACE_ADDR_TYPE. This is used for talking straight to the BMC on the current card. The channel must be IPMI_BMC_CHANNEL.

Messages that are destined to go out on the IPMB bus use the IPMI_IPMB_ADDR_TYPE address type. The format is:

```
struct ipmi_ipmb_addr
{
      int           addr_type;
      short         channel;
      unsigned char slave_addr;
      unsigned char lun;
};
```

The “channel” here is generally zero, but some devices support more than one channel, it corresponds to the channel as defined in the IPMI spec.

26.4 Messages

Messages are defined as:

```
struct ipmi_msg
{
      unsigned char netfn;
      unsigned char lun;
      unsigned char cmd;
      unsigned char *data;
      int           data_len;
};
```

The driver takes care of adding/stripping the header information. The data portion is just the data to be send (do NOT put addressing info here) or the response. Note that the completion code of a response is the first item in “data”, it is not stripped out because that is how all the messages are defined in the spec (and thus makes counting the offsets a little easier :-).

When using the IOCTL interface from userland, you must provide a block of data for “data”, fill it, and set data_len to the length of the block of data, even when receiving messages. Otherwise the driver will have no place to put the message.

Messages coming up from the message handler in kernelland will come in as:

```
struct ipmi_recv_msg
{
      struct list_head link;

      /* The type of message as defined in the "Receive Types"
         defines above. */
      int         recv_type;

      ipmi_user_t      *user;
      struct ipmi_addr addr;
      long             msgid;
      struct ipmi_msg  msg;

      /* Call this when done with the message.  It will presumably free
         the message and do any other necessary cleanup. */
      void (*done)(struct ipmi_recv_msg *msg);

      /* Place-holder for the data, don't make any assumptions about
```

(continues on next page)

(continued from previous page)

```
        the size or existence of this, since it may change. */
     unsigned char   msg_data[IPMI_MAX_MSG_LENGTH];
};
```

You should look at the receive type and handle the message appropriately.

26.5 The Upper Layer Interface (Message Handler)

The upper layer of the interface provides the users with a consistent view of the IPMI interfaces. It allows multiple SMI interfaces to be addressed (because some boards actually have multiple BMCs on them) and the user should not have to care what type of SMI is below them.

26.5.1 Watching For Interfaces

When your code comes up, the IPMI driver may or may not have detected if IPMI devices exist. So you might have to defer your setup until the device is detected, or you might be able to do it immediately. To handle this, and to allow for discovery, you register an SMI watcher with ipmi_smi_watcher_register() to iterate over interfaces and tell you when they come and go.

26.5.2 Creating the User

To use the message handler, you must first create a user using ipmi_create_user. The interface number specifies which SMI you want to connect to, and you must supply callback functions to be called when data comes in. The callback function can run at interrupt level, so be careful using the callbacks. This also allows to you pass in a piece of data, the handler_data, that will be passed back to you on all calls.

Once you are done, call ipmi_destroy_user() to get rid of the user.

From userland, opening the device automatically creates a user, and closing the device automatically destroys the user.

26.5.3 Messaging

To send a message from kernel-land, the ipmi_request_settime() call does pretty much all message handling. Most of the parameter are self-explanatory. However, it takes a "msgid" parameter. This is NOT the sequence number of messages. It is simply a long value that is passed back when the response for the message is returned. You may use it for anything you like.

Responses come back in the function pointed to by the ipmi_recv_hndl field of the "handler" that you passed in to ipmi_create_user(). Remember again, these may be running at interrupt level. Remember to look at the receive type, too.

From userland, you fill out an ipmi_req_t structure and use the IPMICTL_SEND_COMMAND ioctl. For incoming stuff, you can use select() or poll()

to wait for messages to come in. However, you cannot use read() to get them, you must call the IPMICTL_RECEIVE_MSG with the ipmi_recv_t structure to actually get the message. Remember that you must supply a pointer to a block of data in the msg.data field, and you must fill in the msg.data_len field with the size of the data. This gives the receiver a place to actually put the message.

If the message cannot fit into the data you provide, you will get an EMSGSIZE error and the driver will leave the data in the receive queue. If you want to get it and have it truncate the message, us the IPMICTL_RECEIVE_MSG_TRUNC ioctl.

When you send a command (which is defined by the lowest-order bit of the netfn per the IPMI spec) on the IPMB bus, the driver will automatically assign the sequence number to the command and save the command. If the response is not receive in the IPMI-specified 5 seconds, it will generate a response automatically saying the command timed out. If an unsolicited response comes in (if it was after 5 seconds, for instance), that response will be ignored.

In kernelland, after you receive a message and are done with it, you MUST call ipmi_free_recv_msg() on it, or you will leak messages. Note that you should NEVER mess with the "done" field of a message, that is required to properly clean up the message.

Note that when sending, there is an ipmi_request_supply_msgs() call that lets you supply the smi and receive message. This is useful for pieces of code that need to work even if the system is out of buffers (the watchdog timer uses this, for instance). You supply your own buffer and own free routines. This is not recommended for normal use, though, since it is tricky to manage your own buffers.

26.5.4 Events and Incoming Commands

The driver takes care of polling for IPMI events and receiving commands (commands are messages that are not responses, they are commands that other things on the IPMB bus have sent you). To receive these, you must register for them, they will not automatically be sent to you.

To receive events, you must call ipmi_set_gets_events() and set the "val" to non-zero. Any events that have been received by the driver since startup will immediately be delivered to the first user that registers for events. After that, if multiple users are registered for events, they will all receive all events that come in.

For receiving commands, you have to individually register commands you want to receive. Call ipmi_register_for_cmd() and supply the netfn and command name for each command you want to receive. You also specify a bitmask of the channels you want to receive the command from (or use IPMI_CHAN_ALL for all channels if you don't care). Only one user may be registered for each netfn/cmd/channel, but different users may register for different commands, or the same command if the channel bitmasks do not overlap.

From userland, equivalent IOCTLs are provided to do these functions.

26.6 The Lower Layer (SMI) Interface

As mentioned before, multiple SMI interfaces may be registered to the message handler, each of these is assigned an interface number when they register with the message handler. They are generally assigned in the order they register, although if an SMI unregisters and then another one registers, all bets are off.

The ipmi_smi.h defines the interface for management interfaces, see that for more details.

26.7 The SI Driver

The SI driver allows KCS, BT, and SMIC interfaces to be configured in the system. It discovers interfaces through a host of different methods, depending on the system.

You can specify up to four interfaces on the module load line and control some module parameters:

```
modprobe ipmi_si.o type=<type1>,<type2>....
     ports=<port1>,<port2>... addrs=<addr1>,<addr2>...
     irqs=<irq1>,<irq2>...
     regspacings=<sp1>,<sp2>,... regsizes=<size1>,<size2>,...
     regshifts=<shift1>,<shift2>,...
     slave_addrs=<addr1>,<addr2>,...
     force_kipmid=<enable1>,<enable2>,...
     kipmid_max_busy_us=<ustime1>,<ustime2>,...
     unload_when_empty=[0|1]
     trydmi=[0|1] tryacpi=[0|1]
     tryplatform=[0|1] trypci=[0|1]
```

Each of these except try⋯items is a list, the first item for the first interface, second item for the second interface, etc.

The si_type may be either “kcs”, “smic”, or “bt”. If you leave it blank, it defaults to “kcs”.

If you specify addrs as non-zero for an interface, the driver will use the memory address given as the address of the device. This overrides si_ports.

If you specify ports as non-zero for an interface, the driver will use the I/O port given as the device address.

If you specify irqs as non-zero for an interface, the driver will attempt to use the given interrupt for the device.

The other try⋯items disable discovery by their corresponding names. These are all enabled by default, set them to zero to disable them. The tryplatform disables openfirmware.

The next three parameters have to do with register layout. The registers used by the interfaces may not appear at successive locations and they may not be in 8-bit registers. These parameters allow the layout of the data in the registers to be more precisely specified.

The regspacings parameter give the number of bytes between successive register start addresses. For instance, if the regspacing is set to 4 and the start address is 0xca2, then the address for the second register would be 0xca6. This defaults to 1.

The regsizes parameter gives the size of a register, in bytes. The data used by IPMI is 8-bits wide, but it may be inside a larger register. This parameter allows the read and write type to specified. It may be 1, 2, 4, or 8. The default is 1.

Since the register size may be larger than 32 bits, the IPMI data may not be in the lower 8 bits. The regshifts parameter give the amount to shift the data to get to the actual IPMI data.

The slave_addrs specifies the IPMI address of the local BMC. This is usually 0x20 and the driver defaults to that, but in case it's not, it can be specified when the driver starts up.

The force_ipmid parameter forcefully enables (if set to 1) or disables (if set to 0) the kernel IPMI daemon. Normally this is auto-detected by the driver, but systems with broken interrupts might need an enable, or users that don't want the daemon (don't need the performance, don't want the CPU hit) can disable it.

If unload_when_empty is set to 1, the driver will be unloaded if it doesn't find any interfaces or all the interfaces fail to work. The default is one. Setting to 0 is useful with the hotmod, but is obviously only useful for modules.

When compiled into the kernel, the parameters can be specified on the kernel command line as:

```
ipmi_si.type=<type1>,<type2>...
     ipmi_si.ports=<port1>,<port2>... ipmi_si.addrs=<addr1>,<addr2>...
     ipmi_si.irqs=<irq1>,<irq2>...
     ipmi_si.regspacings=<sp1>,<sp2>,...
     ipmi_si.regsizes=<size1>,<size2>,...
     ipmi_si.regshifts=<shift1>,<shift2>,...
     ipmi_si.slave_addrs=<addr1>,<addr2>,...
     ipmi_si.force_kipmid=<enable1>,<enable2>,...
     ipmi_si.kipmid_max_busy_us=<ustime1>,<ustime2>,...
```

It works the same as the module parameters of the same names.

If your IPMI interface does not support interrupts and is a KCS or SMIC interface, the IPMI driver will start a kernel thread for the interface to help speed things up. This is a low-priority kernel thread that constantly polls the IPMI driver while an IPMI operation is in progress. The force_kipmid module parameter will all the user to force this thread on or off. If you force it off and don't have interrupts, the driver will run VERY slowly. Don't blame me, these interfaces suck.

Unfortunately, this thread can use a lot of CPU depending on the interface's performance. This can waste a lot of CPU and cause various issues with detecting idle CPU and using extra power. To avoid this, the kipmid_max_busy_us sets the maximum amount of time, in microseconds, that kipmid will spin before sleeping for a tick. This value sets a balance between performance and CPU waste and needs to be tuned to your needs. Maybe, someday, auto-tuning will be added, but that's not a simple thing and even the auto-tuning would need to be tuned to the user's desired performance.

The driver supports a hot add and remove of interfaces. This way, interfaces can be added or removed after the kernel is up and running. This is done using /sys/modules/ipmi_si/parameters/hotmod, which is a write-only parameter. You write a string to this interface. The string has the format:

```
<op1>[:op2[:op3...]]
```

The "op"s are:

```
add|remove,kcs|bt|smic,mem|i/o,<address>[,<opt1>[,<opt2>[,...]]]
```

You can specify more than one interface on the line. The "opt"s are:

```
rsp=<regspacing>
rsi=<regsize>
rsh=<regshift>
irq=<irq>
ipmb=<ipmb slave addr>
```

and these have the same meanings as discussed above. Note that you can also use this on the kernel command line for a more compact format for specifying an interface. Note that when removing an interface, only the first three parameters (si type, address type, and address) are used for the comparison. Any options are ignored for removing.

26.8 The SMBus Driver (SSIF)

The SMBus driver allows up to 4 SMBus devices to be configured in the system. By default, the driver will only register with something it finds in DMI or ACPI tables. You can change this at module load time (for a module) with:

```
modprobe ipmi_ssif.o
      addr=<i2caddr1>[,<i2caddr2>[,...]]
      adapter=<adapter1>[,<adapter2>[...]]
      dbg=<flags1>,<flags2>...
      slave_addrs=<addr1>,<addr2>,...
      tryacpi=[0|1] trydmi=[0|1]
      [dbg_probe=1]
```

The addresses are normal I2C addresses. The adapter is the string name of the adapter, as shown in /sys/class/i2c-adapter/i2c-<n>/name. It is NOT i2c-<n> itself. Also, the comparison is done ignoring spaces, so if the name is "This is an I2C chip" you can say adapter_name=ThisisanI2cchip. This is because it's hard to pass in spaces in kernel parameters.

The debug flags are bit flags for each BMC found, they are: IPMI messages: 1, driver state: 2, timing: 4, I2C probe: 8

The tryxxx parameters can be used to disable detecting interfaces from various sources.

Setting dbg_probe to 1 will enable debugging of the probing and detection process for BMCs on the SMBusses.

The slave_addrs specifies the IPMI address of the local BMC. This is usually 0x20 and the driver defaults to that, but in case it's not, it can be specified when the driver starts up.

Discovering the IPMI compliant BMC on the SMBus can cause devices on the I2C bus to fail. The SMBus driver writes a "Get Device ID" IPMI message as a block write to the I2C bus and waits for a response. This action can be detrimental to some I2C devices. It is highly recommended that the known I2C address be given to the SMBus driver in the smb_addr parameter unless you have DMI or ACPI data to tell the driver what to use.

When compiled into the kernel, the addresses can be specified on the kernel command line as:

```
ipmb_ssif.addr=<i2caddr1>[,<i2caddr2>[...]]
      ipmi_ssif.adapter=<adapter1>[,<adapter2>[...]]
      ipmi_ssif.dbg=<flags1>[,<flags2>[...]]
      ipmi_ssif.dbg_probe=1
      ipmi_ssif.slave_addrs=<addr1>[,<addr2>[...]]
      ipmi_ssif.tryacpi=[0|1] ipmi_ssif.trydmi=[0|1]
```

These are the same options as on the module command line.

The I2C driver does not support non-blocking access or polling, so this driver cannod to IPMI panic events, extend the watchdog at panic time, or other panic-related IPMI functions without special kernel patches and driver modifications. You can get those at the openipmi web page.

The driver supports a hot add and remove of interfaces through the I2C sysfs interface.

26.9 Other Pieces

26.10 Get the detailed info related with the IPMI device

Some users need more detailed information about a device, like where the address came from or the raw base device for the IPMI interface. You can use the IPMI smi_watcher to catch the IPMI interfaces as they come or go, and to grab the information, you can use the function ipmi_get_smi_info(), which returns the following structure:

```
struct ipmi_smi_info {
      enum ipmi_addr_src addr_src;
      struct device *dev;
      union {
              struct {
                      void *acpi_handle;
              } acpi_info;
      } addr_info;
};
```

Currently special info for only for SI_ACPI address sources is returned. Others may be added as necessary.

Note that the dev pointer is included in the above structure, and assuming ipmi_smi_get_info returns success, you must call put_device on the dev pointer.

26.11 Watchdog

A watchdog timer is provided that implements the Linux-standard watchdog timer interface. It has three module parameters that can be used to control it:

```
modprobe ipmi_watchdog timeout=<t> pretimeout=<t> action=<action type>
    preaction=<preaction type> preop=<preop type> start_now=x
    nowayout=x ifnum_to_use=n panic_wdt_timeout=<t>
```

ifnum_to_use specifies which interface the watchdog timer should use. The default is -1, which means to pick the first one registered.

The timeout is the number of seconds to the action, and the pretimeout is the amount of seconds before the reset that the pre-timeout panic will occur (if pretimeout is zero, then pretimeout will not be enabled). Note that the pretimeout is the time before the final timeout. So if the timeout is 50 seconds and the pretimeout is 10 seconds, then the pretimeout will occur in 40 second (10 seconds before the timeout). The panic_wdt_timeout is the value of timeout which is set on kernel panic, in order to let actions such as kdump to occur during panic.

The action may be "reset", "power_cycle", or "power_off", and specifies what to do when the timer times out, and defaults to "reset".

The preaction may be "pre_smi" for an indication through the SMI interface, "pre_int" for an indication through the SMI with an interrupts, and "pre_nmi" for a NMI on a preaction. This is how the driver is informed of the pretimeout.

The preop may be set to "preop_none" for no operation on a pretimeout, "preop_panic" to set the preoperation to panic, or "preop_give_data" to provide data to read from the watchdog device when the pretimeout occurs. A "pre_nmi" setting CANNOT be used with "preop_give_data" because you can't do data operations from an NMI.

When preop is set to "preop_give_data", one byte comes ready to read on the device when the pretimeout occurs. Select and fasync work on the device, as well.

If start_now is set to 1, the watchdog timer will start running as soon as the driver is loaded.

If nowayout is set to 1, the watchdog timer will not stop when the watchdog device is closed. The default value of nowayout is true if the CONFIG_WATCHDOG_NOWAYOUT option is enabled, or false if not.

When compiled into the kernel, the kernel command line is available for configuring the watchdog:

```
ipmi_watchdog.timeout=<t> ipmi_watchdog.pretimeout=<t>
      ipmi_watchdog.action=<action type>
      ipmi_watchdog.preaction=<preaction type>
      ipmi_watchdog.preop=<preop type>
      ipmi_watchdog.start_now=x
```

(continues on next page)

(continued from previous page)

```
    ipmi_watchdog.nowayout=x
    ipmi_watchdog.panic_wdt_timeout=<t>
```

The options are the same as the module parameter options.

The watchdog will panic and start a 120 second reset timeout if it gets a pre-action. During a panic or a reboot, the watchdog will start a 120 timer if it is running to make sure the reboot occurs.

Note that if you use the NMI preaction for the watchdog, you MUST NOT use the nmi watchdog. There is no reasonable way to tell if an NMI comes from the IPMI controller, so it must assume that if it gets an otherwise unhandled NMI, it must be from IPMI and it will panic immediately.

Once you open the watchdog timer, you must write a 'V' character to the device to close it, or the timer will not stop. This is a new semantic for the driver, but makes it consistent with the rest of the watchdog drivers in Linux.

26.12 Panic Timeouts

The OpenIPMI driver supports the ability to put semi-custom and custom events in the system event log if a panic occurs. if you enable the 'Generate a panic event to all BMCs on a panic' option, you will get one event on a panic in a standard IPMI event format. If you enable the 'Generate OEM events containing the panic string' option, you will also get a bunch of OEM events holding the panic string.

The field settings of the events are:

- Generator ID: 0x21 (kernel)
- EvM Rev: 0x03 (this event is formatting in IPMI 1.0 format)
- Sensor Type: 0x20 (OS critical stop sensor)
- Sensor #: The first byte of the panic string (0 if no panic string)
- Event Dir | Event Type: 0x6f (Assertion, sensor-specific event info)
- Event Data 1: 0xa1 (Runtime stop in OEM bytes 2 and 3)
- Event data 2: second byte of panic string
- Event data 3: third byte of panic string

See the IPMI spec for the details of the event layout. This event is always sent to the local management controller. It will handle routing the message to the right place

Other OEM events have the following format:

- Record ID (bytes 0-1): Set by the SEL.
- Record type (byte 2): 0xf0 (OEM non-timestamped)
- byte 3: The slave address of the card saving the panic

- byte 4: A sequence number (starting at zero) The rest of the bytes (11 bytes) are the panic string. If the panic string is longer than 11 bytes, multiple messages will be sent with increasing sequence numbers.

Because you cannot send OEM events using the standard interface, this function will attempt to find an SEL and add the events there. It will first query the capabilities of the local management controller. If it has an SEL, then they will be stored in the SEL of the local management controller. If not, and the local management controller is an event generator, the event receiver from the local management controller will be queried and the events sent to the SEL on that device. Otherwise, the events go nowhere since there is nowhere to send them.

26.13 Poweroff

If the poweroff capability is selected, the IPMI driver will install a shutdown function into the standard poweroff function pointer. This is in the ipmi_poweroff module. When the system requests a powerdown, it will send the proper IPMI commands to do this. This is supported on several platforms.

There is a module parameter named "poweroff_powercycle" that may either be zero (do a power down) or non-zero (do a power cycle, power the system off, then power it on in a few seconds). Setting ipmi_poweroff.poweroff_control=x will do the same thing on the kernel command line. The parameter is also available via the proc filesystem in /proc/sys/dev/ipmi/poweroff_powercycle. Note that if the system does not support power cycling, it will always do the power off.

The "ifnum_to_use" parameter specifies which interface the poweroff code should use. The default is -1, which means to pick the first one registered.

Note that if you have ACPI enabled, the system will prefer using ACPI to power off.

CHAPTER

TWENTYSEVEN

I3C SUBSYSTEM

27.1 I3C protocol

27.1.1 Disclaimer

This chapter will focus on aspects that matter to software developers. For everything hardware related (like how things are transmitted on the bus, how collisions are prevented, ⋯) please have a look at the I3C specification.

This document is just a brief introduction to the I3C protocol and the concepts it brings to the table. If you need more information, please refer to the MIPI I3C specification (can be downloaded here http://resources.mipi.org/mipi-i3c-v1-download).

27.1.2 Introduction

The I3C (pronounced ‘eye-three-see’) is a MIPI standardized protocol designed to overcome I2C limitations (limited speed, external signals needed for interrupts, no automatic detection of the devices connected to the bus, ⋯) while remaining power-efficient.

27.1.3 I3C Bus

An I3C bus is made of several I3C devices and possibly some I2C devices as well, but let’s focus on I3C devices for now.

An I3C device on the I3C bus can have one of the following roles:

- Master: the device is driving the bus. It’s the one in charge of initiating transactions or deciding who is allowed to talk on the bus (slave generated events are possible in I3C, see below).
- Slave: the device acts as a slave, and is not able to send frames to another slave on the bus. The device can still send events to the master on its own initiative if the master allowed it.

I3C is a multi-master protocol, so there might be several masters on a bus, though only one device can act as a master at a given time. In order to gain bus ownership, a master has to follow a specific procedure.

Each device on the I3C bus has to be assigned a dynamic address to be able to communicate. Until this is done, the device should only respond to a limited set of commands. If it has a static address (also called legacy I2C address), the device can reply to I2C transfers.

In addition to these per-device addresses, the protocol defines a broadcast address in order to address all devices on the bus.

Once a dynamic address has been assigned to a device, this address will be used for any direct communication with the device. Note that even after being assigned a dynamic address, the device should still process broadcast messages.

27.1.4 I3C Device discovery

The I3C protocol defines a mechanism to automatically discover devices present on the bus, their capabilities and the functionalities they provide. In this regard I3C is closer to a discoverable bus like USB than it is to I2C or SPI.

The discovery mechanism is called DAA (Dynamic Address Assignment), because it not only discovers devices but also assigns them a dynamic address.

During DAA, each I3C device reports 3 important things:

- BCR: Bus Characteristic Register. This 8-bit register describes the device bus related capabilities
- DCR: Device Characteristic Register. This 8-bit register describes the functionalities provided by the device
- Provisional ID: A 48-bit unique identifier. On a given bus there should be no Provisional ID collision, otherwise the discovery mechanism may fail.

27.1.5 I3C slave events

The I3C protocol allows slaves to generate events on their own, and thus allows them to take temporary control of the bus.

This mechanism is called IBI for In Band Interrupts, and as stated in the name, it allows devices to generate interrupts without requiring an external signal.

During DAA, each device on the bus has been assigned an address, and this address will serve as a priority identifier to determine who wins if 2 different devices are generating an interrupt at the same moment on the bus (the lower the dynamic address the higher the priority).

Masters are allowed to inhibit interrupts if they want to. This inhibition request can be broadcast (applies to all devices) or sent to a specific device.

27.1.6 I3C Hot-Join

The Hot-Join mechanism is similar to USB hotplug. This mechanism allows slaves to join the bus after it has been initialized by the master.

This covers the following use cases:

- the device is not powered when the bus is probed
- the device is hotplugged on the bus through an extension board

This mechanism is relying on slave events to inform the master that a new device joined the bus and is waiting for a dynamic address.

The master is then free to address the request as it wishes: ignore it or assign a dynamic address to the slave.

27.1.7 I3C transfer types

If you omit SMBus (which is just a standardization on how to access registers exposed by I2C devices), I2C has only one transfer type.

I3C defines 3 different classes of transfer in addition to I2C transfers which are here for backward compatibility with I2C devices.

I3C CCC commands

CCC (Common Command Code) commands are meant to be used for anything that is related to bus management and all features that are common to a set of devices.

CCC commands contain an 8-bit CCC ID describing the command that is executed. The MSB of this ID specifies whether this is a broadcast command (bit7 = 0) or a unicast one (bit7 = 1).

The command ID can be followed by a payload. Depending on the command, this payload is either sent by the master sending the command (write CCC command), or sent by the slave receiving the command (read CCC command). Of course, read accesses only apply to unicast commands. Note that, when sending a CCC command to a specific device, the device address is passed in the first byte of the payload.

The payload length is not explicitly passed on the bus, and should be extracted from the CCC ID.

Note that vendors can use a dedicated range of CCC IDs for their own commands (0x61-0x7f and 0xe0-0xef).

I3C Private SDR transfers

Private SDR (Single Data Rate) transfers should be used for anything that is device specific and does not require high transfer speed.

It is the equivalent of I2C transfers but in the I3C world. Each transfer is passed the device address (dynamic address assigned during DAA), a payload and a direction.

The only difference with I2C is that the transfer is much faster (typical clock frequency is 12.5MHz).

I3C HDR commands

HDR commands should be used for anything that is device specific and requires high transfer speed.

The first thing attached to an HDR command is the HDR mode. There are currently 3 different modes defined by the I3C specification (refer to the specification for more details):

- HDR-DDR: Double Data Rate mode
- HDR-TSP: Ternary Symbol Pure. Only usable on busses with no I2C devices
- HDR-TSL: Ternary Symbol Legacy. Usable on busses with I2C devices

When sending an HDR command, the whole bus has to enter HDR mode, which is done using a broadcast CCC command. Once the bus has entered a specific HDR mode, the master sends the HDR command. An HDR command is made of:

- one 16-bits command word in big endian
- N 16-bits data words in big endian

Those words may be wrapped with specific preambles/post-ambles which depend on the chosen HDR mode and are detailed here (see the specification for more details).

The 16-bits command word is made of:

- bit[15]: direction bit, read is 1, write is 0
- bit[14:8]: command code. Identifies the command being executed, the amount of data words and their meaning
- bit[7:1]: I3C address of the device this command is addressed to
- bit[0]: reserved/parity-bit

27.1.8 Backward compatibility with I2C devices

The I3C protocol has been designed to be backward compatible with I2C devices. This backward compatibility allows one to connect a mix of I2C and I3C devices on the same bus, though, in order to be really efficient, I2C devices should be equipped with 50 ns spike filters.

I2C devices can't be discovered like I3C ones and have to be statically declared. In order to let the master know what these devices are capable of (both in terms of bus related limitations and functionalities), the software has to provide some information, which is done through the LVR (Legacy I2C Virtual Register).

27.2 I3C device driver API

enum **i3c_error_code**
I3C error codes

Constants

I3C_ERROR_UNKNOWN unknown error, usually means the error is not I3C related

I3C_ERROR_M0 M0 error

I3C_ERROR_M1 M1 error

I3C_ERROR_M2 M2 error

Description

These are the standard error codes as defined by the I3C specification. When -EIO is returned by the `i3c_device_do_priv_xfers()` or i3c_device_send_hdr_cmds() one can check the error code in `struct i3c_priv_xfer.err` or `struct i3c_hdr_cmd`.err to get a better idea of what went wrong.

enum **i3c_hdr_mode**
HDR mode ids

Constants

I3C_HDR_DDR DDR mode

I3C_HDR_TSP TSP mode

I3C_HDR_TSL TSL mode

struct **i3c_priv_xfer**
I3C SDR private transfer

Definition

```
struct i3c_priv_xfer {
  u8 rnw;
  u16 len;
  union {
    void *in;
    const void *out;
  } data;
```

(continues on next page)

(continued from previous page)

```
  enum i3c_error_code err;
};
```

Members

rnw encodes the transfer direction. true for a read, false for a write

len transfer length in bytes of the transfer

data input/output buffer

data.in input buffer. Must point to a DMA-able buffer

data.out output buffer. Must point to a DMA-able buffer

err I3C error code

enum **i3c_dcr**
I3C DCR values

Constants

I3C_DCR_GENERIC_DEVICE generic I3C device

struct **i3c_device_info**
I3C device information

Definition

```
struct i3c_device_info {
  u64 pid;
  u8 bcr;
  u8 dcr;
  u8 static_addr;
  u8 dyn_addr;
  u8 hdr_cap;
  u8 max_read_ds;
  u8 max_write_ds;
  u8 max_ibi_len;
  u32 max_read_turnaround;
  u16 max_read_len;
  u16 max_write_len;
};
```

Members

pid Provisional ID

bcr Bus Characteristic Register

dcr Device Characteristic Register

static_addr static/I2C address

dyn_addr dynamic address

hdr_cap supported HDR modes

max_read_ds max read speed information

max_write_ds max write speed information

max_ibi_len max IBI payload length

max_read_turnaround max read turn-around time in micro-seconds

max_read_len max private SDR read length in bytes

max_write_len max private SDR write length in bytes

Description

These are all basic information that should be advertised by an I3C device. Some of them are optional depending on the device type and device capabilities. For each I3C slave attached to a master with `i3c_master_add_i3c_dev_locked()`, the core will send the relevant CCC command to retrieve these data.

struct **i3c_driver**
I3C device driver

Definition

```
struct i3c_driver {
  struct device_driver driver;
  int (*probe)(struct i3c_device *dev);
  int (*remove)(struct i3c_device *dev);
  const struct i3c_device_id *id_table;
};
```

Members

driver inherit from device_driver

probe I3C device probe method

remove I3C device remove method

id_table I3C device match table. Will be used by the framework to decide which device to bind to this driver

module_i3c_driver(__drv)
Register a module providing an I3C driver

Parameters

__drv the I3C driver to register

Description

Provide generic init/exit functions that simply register/unregister an I3C driver. Should be used by any driver that does not require extra init/cleanup steps.

int **i3c_i2c_driver_register**(struct i3c_driver * i3cdrv, struct i2c_driver * i2cdrv)
Register an i2c and an i3c driver

Parameters

struct i3c_driver * i3cdrv the I3C driver to register

struct i2c_driver * i2cdrv the I2C driver to register

Description

This function registers both **i2cdev** and **i3cdev**, and fails if one of these registrations fails. This is mainly useful for devices that support both I2C and I3C modes. Note that when CONFIG_I3C is not enabled, this function only registers the I2C driver.

Return

0 if both registrations succeeds, a negative error code otherwise.

void **i3c_i2c_driver_unregister**(struct i3c_driver * i3cdrv, struct i2c_driver * i2cdrv)
Unregister an i2c and an i3c driver

Parameters

struct i3c_driver * i3cdrv the I3C driver to register

struct i2c_driver * i2cdrv the I2C driver to register

Description

This function unregisters both **i3cdrv** and **i2cdrv**. Note that when CONFIG_I3C is not enabled, this function only unregisters the **i2cdrv**.

module_i3c_i2c_driver(__i3cdrv, __i2cdrv)
Register a module providing an I3C and an I2C driver

Parameters

__i3cdrv the I3C driver to register

__i2cdrv the I3C driver to register

Description

Provide generic init/exit functions that simply register/unregister an I3C and an I2C driver. This macro can be used even if CONFIG_I3C is disabled, in this case, only the I2C driver will be registered. Should be used by any driver that does not require extra init/cleanup steps.

struct **i3c_ibi_setup**
IBI setup object

Definition

```
struct i3c_ibi_setup {
  unsigned int max_payload_len;
  unsigned int num_slots;
  void (*handler)(struct i3c_device *dev, const struct i3c_ibi_payload␣
↪*payload);
};
```

Members

max_payload_len maximum length of the payload associated to an IBI. If one IBI appears to have a payload that is bigger than this number, the IBI will be rejected.

num_slots number of pre-allocated IBI slots. This should be chosen so that the system never runs out of IBI slots, otherwise you' ll lose IBIs.

handler IBI handler, every time an IBI is received. This handler is called in a workqueue context. It is allowed to sleep and send new messages on the bus, though it's recommended to keep the processing done there as fast as possible to avoid delaying processing of other queued on the same workqueue.

Description

Temporary structure used to pass information to `i3c_device_request_ibi()`. This object can be allocated on the stack since `i3c_device_request_ibi()` copies every bit of information and do not use it after `i3c_device_request_ibi()` has returned.

int **i3c_device_do_priv_xfers**(struct i3c_device * dev, struct i3c_priv_xfer * xfers, int nxfers)
do I3C SDR private transfers directed to a specific device

Parameters

struct i3c_device * dev device with which the transfers should be done

struct i3c_priv_xfer * xfers array of transfers

int nxfers number of transfers

Description

Initiate one or several private SDR transfers with **dev**.

This function can sleep and thus cannot be called in atomic context.

Return

0 in case of success, a negative error core otherwise.

void **i3c_device_get_info**(struct i3c_device * dev, struct i3c_device_info * info)
get I3C device information

Parameters

struct i3c_device * dev device we want information on

struct i3c_device_info * info the information object to fill in

Description

Retrieve I3C dev info.

int **i3c_device_disable_ibi**(struct i3c_device * dev)
Disable IBIs coming from a specific device

Parameters

struct i3c_device * dev device on which IBIs should be disabled

Description

This function disable IBIs coming from a specific device and wait for all pending IBIs to be processed.

Return

0 in case of success, a negative error core otherwise.

int **i3c_device_enable_ibi**(struct i3c_device * dev)
Enable IBIs coming from a specific device

Parameters

struct i3c_device * dev device on which IBIs should be enabled

Description

This function enable IBIs coming from a specific device and wait for all pending IBIs to be processed. This should be called on a device where `i3c_device_request_ibi()` has succeeded.

Note that IBIs from this device might be received before this function returns to its caller.

Return

0 in case of success, a negative error core otherwise.

int **i3c_device_request_ibi**(struct i3c_device * dev, const struct i3c_ibi_setup * req)
Request an IBI

Parameters

struct i3c_device * dev device for which we should enable IBIs

const struct i3c_ibi_setup * req setup requested for this IBI

Description

This function is responsible for pre-allocating all resources needed to process IBIs coming from **dev**. When this function returns, the IBI is not enabled until `i3c_device_enable_ibi()` is called.

Return

0 in case of success, a negative error core otherwise.

void **i3c_device_free_ibi**(struct i3c_device * dev)
Free all resources needed for IBI handling

Parameters

struct i3c_device * dev device on which you want to release IBI resources

Description

This function is responsible for de-allocating resources previously allocated by `i3c_device_request_ibi()`. It should be called after disabling IBIs with `i3c_device_disable_ibi()`.

struct device * **i3cdev_to_dev**(struct i3c_device * i3cdev)
Returns the device embedded in **i3cdev**

Parameters

struct i3c_device * i3cdev I3C device

Return

a pointer to a device object.

struct i3c_device * **dev_to_i3cdev**(struct device * dev)
Returns the I3C device containing **dev**

Parameters

struct device * dev device object

Return

a pointer to an I3C device object.

const struct i3c_device_id * **i3c_device_match_id**(struct i3c_device * i3cdev, const struct i3c_device_id * id_table)
Returns the i3c_device_id entry matching **i3cdev**

Parameters

struct i3c_device * i3cdev I3C device

const struct i3c_device_id * id_table I3C device match table

Return

a pointer to an i3c_device_id object or NULL if there's no match.

int **i3c_driver_register_with_owner**(struct i3c_driver * drv, struct module * owner)
register an I3C device driver

Parameters

struct i3c_driver * drv driver to register

struct module * owner module that owns this driver

Description

Register **drv** to the core.

Return

0 in case of success, a negative error core otherwise.

void **i3c_driver_unregister**(struct i3c_driver * drv)
unregister an I3C device driver

Parameters

struct i3c_driver * drv driver to unregister

Description

Unregister **drv**.

27.3 I3C master controller driver API

void **i3c_bus_maintenance_lock**(struct i3c_bus * bus)
Lock the bus for a maintenance operation

Parameters

struct i3c_bus * bus I3C bus to take the lock on

Description

This function takes the bus lock so that no other operations can occur on the bus. This is needed for all kind of bus maintenance operation, like - enabling/disabling slave events - re-triggering DAA - changing the dynamic address of a device - relinquishing mastership - ⋯

The reason for this kind of locking is that we don't want drivers and core logic to rely on I3C device information that could be changed behind their back.

void **i3c_bus_maintenance_unlock**(struct i3c_bus * bus)
Release the bus lock after a maintenance operation

Parameters

struct i3c_bus * bus I3C bus to release the lock on

Description

Should be called when the bus maintenance operation is done. See `i3c_bus_maintenance_lock()` for more details on what these maintenance operations are.

void **i3c_bus_normaluse_lock**(struct i3c_bus * bus)
Lock the bus for a normal operation

Parameters

struct i3c_bus * bus I3C bus to take the lock on

Description

This function takes the bus lock for any operation that is not a maintenance operation (see `i3c_bus_maintenance_lock()` for a non-exhaustive list of maintenance operations). Basically all communications with I3C devices are normal operations (HDR, SDR transfers or CCC commands that do not change bus state or I3C dynamic address).

Note that this lock is not guaranteeing serialization of normal operations. In other words, transfer requests passed to the I3C master can be submitted in parallel and I3C master drivers have to use their own locking to make sure two different communications are not inter-mixed, or access to the output/input queue is not done while the engine is busy.

void **i3c_bus_normaluse_unlock**(struct i3c_bus * bus)
Release the bus lock after a normal operation

Parameters

struct i3c_bus * bus I3C bus to release the lock on

Description

Should be called when a normal operation is done. See `i3c_bus_normaluse_lock()` for more details on what these normal operations are.

int **i3c_master_get_free_addr**(struct i3c_master_controller * master, u8 start_addr)
get a free address on the bus

Parameters

struct i3c_master_controller * master I3C master object

u8 start_addr where to start searching

Description

This function must be called with the bus lock held in write mode.

Return

the first free address starting at **start_addr** (included) or -ENOMEM if there's no more address available.

int **i3c_master_entdaa_locked**(struct i3c_master_controller * master)
start a DAA (Dynamic Address Assignment) procedure

Parameters

struct i3c_master_controller * master master used to send frames on the bus

Description

Send a ENTDAA CCC command to start a DAA procedure.

Note that this function only sends the ENTDAA CCC command, all the logic behind dynamic address assignment has to be handled in the I3C master driver.

This function must be called with the bus lock held in write mode.

Return

0 in case of success, a positive I3C error code if the error is one of the official Mx error codes, and a negative error code otherwise.

int **i3c_master_disec_locked**(struct i3c_master_controller * master, u8 addr, u8 evts)
send a DISEC CCC command

Parameters

struct i3c_master_controller * master master used to send frames on the bus

u8 addr a valid I3C slave address or `I3C_BROADCAST_ADDR`

u8 evts events to disable

Description

Send a DISEC CCC command to disable some or all events coming from a specific slave, or all devices if **addr** is `I3C_BROADCAST_ADDR`.

This function must be called with the bus lock held in write mode.

Return

0 in case of success, a positive I3C error code if the error is one of the official Mx error codes, and a negative error code otherwise.

int **i3c_master_enec_locked**(struct i3c_master_controller * master, u8 addr, u8 evts)
send an ENEC CCC command

Parameters

struct i3c_master_controller * master master used to send frames on the bus

u8 addr a valid I3C slave address or `I3C_BROADCAST_ADDR`

u8 evts events to disable

Description

Sends an ENEC CCC command to enable some or all events coming from a specific slave, or all devices if **addr** is `I3C_BROADCAST_ADDR`.

This function must be called with the bus lock held in write mode.

Return

0 in case of success, a positive I3C error code if the error is one of the official Mx error codes, and a negative error code otherwise.

int **i3c_master_defslvs_locked**(struct i3c_master_controller * master)
send a DEFSLVS CCC command

Parameters

struct i3c_master_controller * master master used to send frames on the bus

Description

Send a DEFSLVS CCC command containing all the devices known to the **master**. This is useful when you have secondary masters on the bus to propagate device information.

This should be called after all I3C devices have been discovered (in other words, after the DAA procedure has finished) and instantiated in `i3c_master_controller_ops->bus_init()`. It should also be called if a master ACKed an Hot-Join request and assigned a dynamic address to the device joining the bus.

This function must be called with the bus lock held in write mode.

Return

0 in case of success, a positive I3C error code if the error is one of the official Mx error codes, and a negative error code otherwise.

int **i3c_master_do_daa**(struct i3c_master_controller * master)
do a DAA (Dynamic Address Assignment)

Parameters

struct i3c_master_controller * master master doing the DAA

Description

This function is instantiating an I3C device object and adding it to the I3C device list. All device information are automatically retrieved using standard CCC commands.

The I3C device object is returned in case the master wants to attach private data to it using `i3c_dev_set_master_data()`.

This function must be called with the bus lock held in write mode.

Return

a 0 in case of success, an negative error code otherwise.

int **i3c_master_set_info**(struct i3c_master_controller * master, const struct i3c_device_info * info)
set master device information

Parameters

struct i3c_master_controller * master master used to send frames on the bus

const struct i3c_device_info * info I3C device information

Description

Set master device info. This should be called from `i3c_master_controller_ops->bus_init()`.

Not all `i3c_device_info` fields are meaningful for a master device. Here is a list of fields that should be properly filled:

- `i3c_device_info->dyn_addr`
- `i3c_device_info->bcr`
- `i3c_device_info->dcr`
- `i3c_device_info->pid`
- `i3c_device_info->hdr_cap` if `I3C_BCR_HDR_CAP` bit is set in `i3c_device_info->bcr`

This function must be called with the bus lock held in maintenance mode.

Return

0 if **info** contains valid information (not every piece of information can be checked, but we can at least make sure **info->dyn_addr** and **info->bcr** are correct), -EINVAL otherwise.

int **i3c_master_bus_init**(struct i3c_master_controller * master)
initialize an I3C bus

Parameters

struct i3c_master_controller * master main master initializing the bus

Description

This function is following all initialisation steps described in the I3C specification:

1. Attach I2C and statically defined I3C devs to the master so that the master can fill its internal device table appropriately
2. Call `i3c_master_controller_ops->bus_init()` method to initialize the master controller. That's usually where the bus mode is selected (pure bus or mixed fast/slow bus)
3. Instruct all devices on the bus to drop their dynamic address. This is particularly important when the bus was previously configured by someone else (for example the bootloader)
4. Disable all slave events.
5. Pre-assign dynamic addresses requested by the FW with SETDASA for I3C devices that have a static address
6. Do a DAA (Dynamic Address Assignment) to assign dynamic addresses to all remaining I3C devices

Once this is done, all I3C and I2C devices should be usable.

Return

a 0 in case of success, an negative error code otherwise.

int **i3c_master_add_i3c_dev_locked**(struct i3c_master_controller * master, u8 addr)

add an I3C slave to the bus

Parameters

struct i3c_master_controller * master master used to send frames on the bus

u8 addr I3C slave dynamic address assigned to the device

Description

This function is instantiating an I3C device object and adding it to the I3C device list. All device information are automatically retrieved using standard CCC commands.

The I3C device object is returned in case the master wants to attach private data to it using `i3c_dev_set_master_data()`.

This function must be called with the bus lock held in write mode.

Return

a 0 in case of success, an negative error code otherwise.

void **i3c_master_queue_ibi**(struct i3c_dev_desc * dev, struct i3c_ibi_slot * slot)

Queue an IBI

Parameters

struct i3c_dev_desc * dev the device this IBI is coming from

struct i3c_ibi_slot * slot the IBI slot used to store the payload

Description

Queue an IBI to the controller workqueue. The IBI handler attached to the dev will be called from a workqueue context.

void **i3c_generic_ibi_free_pool**(struct i3c_generic_ibi_pool * pool)
Free a generic IBI pool

Parameters

struct i3c_generic_ibi_pool * pool the IBI pool to free

Description

Free all IBI slots allated by a generic IBI pool.

struct i3c_generic_ibi_pool * **i3c_generic_ibi_alloc_pool**(struct i3c_dev_desc * dev, const struct i3c_ibi_setup * req)
Create a generic IBI pool

Parameters

struct i3c_dev_desc * dev the device this pool will be used for

const struct i3c_ibi_setup * req IBI setup request describing what the device driver expects

Description

Create a generic IBI pool based on the information provided in **req**.

Return

a valid IBI pool in case of success, an ERR_PTR() otherwise.

struct i3c_ibi_slot * **i3c_generic_ibi_get_free_slot**(struct i3c_generic_ibi_pool * pool)
Get a free slot from a generic IBI pool

Parameters

struct i3c_generic_ibi_pool * pool the pool to query an IBI slot on

Description

Search for a free slot in a generic IBI pool. The slot should be returned to the pool using `i3c_generic_ibi_recycle_slot()` when it's no longer needed.

Return

a pointer to a free slot, or NULL if there's no free slot available.

void **i3c_generic_ibi_recycle_slot**(struct i3c_generic_ibi_pool * pool, struct i3c_ibi_slot * s)
Return a slot to a generic IBI pool

Parameters

struct i3c_generic_ibi_pool * pool the pool to return the IBI slot to

struct i3c_ibi_slot * s IBI slot to recycle

Description

Add an IBI slot back to its generic IBI pool. Should be called from the master driver struct_master_controller_ops->recycle_ibi() method.

int **i3c_master_register**(struct i3c_master_controller * master, struct device * parent, const struct i3c_master_controller_ops * ops, bool secondary)
register an I3C master

Parameters

struct i3c_master_controller * master master used to send frames on the bus

struct device * parent the parent device (the one that provides this I3C master controller)

const struct i3c_master_controller_ops * ops the master controller operations

bool secondary true if you are registering a secondary master. Will return -ENOTSUPP if set to true since secondary masters are not yet supported

Description

This function takes care of everything for you:

- creates and initializes the I3C bus
- populates the bus with static I2C devs if **parent->of_node** is not NULL
- registers all I3C devices added by the controller during bus initialization
- registers the I2C adapter and all I2C devices

Return

0 in case of success, a negative error code otherwise.

int **i3c_master_unregister**(struct i3c_master_controller * master)
unregister an I3C master

Parameters

struct i3c_master_controller * master master used to send frames on the bus

Description

Basically undo everything done in `i3c_master_register()`.

Return

0 in case of success, a negative error code otherwise.

struct **i3c_i2c_dev_desc**
Common part of the I3C/I2C device descriptor

Definition

```
struct i3c_i2c_dev_desc {
  struct list_head node;
  struct i3c_master_controller *master;
  void *master_priv;
};
```

Members

node node element used to insert the slot into the I2C or I3C device list

master I3C master that instantiated this device. Will be used to do I2C/I3C transfers

master_priv master private data assigned to the device. Can be used to add master specific information

Description

This structure is describing common I3C/I2C dev information.

struct **i2c_dev_boardinfo**
I2C device board information

Definition

```
struct i2c_dev_boardinfo {
  struct list_head node;
  struct i2c_board_info base;
  u8 lvr;
};
```

Members

node used to insert the boardinfo object in the I2C boardinfo list

base regular I2C board information

lvr LVR (Legacy Virtual Register) needed by the I3C core to know about the I2C device limitations

Description

This structure is used to attach board-level information to an I2C device. Each I2C device connected on the I3C bus should have one.

struct **i2c_dev_desc**
I2C device descriptor

Definition

```
struct i2c_dev_desc {
  struct i3c_i2c_dev_desc common;
  const struct i2c_dev_boardinfo *boardinfo;
  struct i2c_client *dev;
  u16 addr;
  u8 lvr;
};
```

Members

common common part of the I2C device descriptor

boardinfo pointer to the boardinfo attached to this I2C device

dev I2C device object registered to the I2C framework

addr I2C device address

lvr LVR (Legacy Virtual Register) needed by the I3C core to know about the I2C device limitations

Description

Each I2C device connected on the bus will have an i2c_dev_desc. This object is created by the core and later attached to the controller using `struct_i3c_master_controller->ops->`attach_i2c_dev().

`struct_i2c_dev_desc` is the internal representation of an I2C device connected on an I3C bus. This object is also passed to all `struct_i3c_master_controller_ops` hooks.

struct **i3c_ibi_slot**
I3C IBI (In-Band Interrupt) slot

Definition

```
struct i3c_ibi_slot {
  struct work_struct work;
  struct i3c_dev_desc *dev;
  unsigned int len;
  void *data;
};
```

Members

work work associated to this slot. The IBI handler will be called from there

dev the I3C device that has generated this IBI

len length of the payload associated to this IBI

data payload buffer

Description

An IBI slot is an object pre-allocated by the controller and used when an IBI comes in. Every time an IBI comes in, the I3C master driver should find a free IBI slot in its IBI slot pool, retrieve the IBI payload and queue the IBI using `i3c_master_queue_ibi()`.

How IBI slots are allocated is left to the I3C master driver, though, for simple kmalloc-based allocation, the generic IBI slot pool can be used.

struct **i3c_device_ibi_info**
IBI information attached to a specific device

Definition

```
struct i3c_device_ibi_info {
  struct completion all_ibis_handled;
  atomic_t pending_ibis;
  unsigned int max_payload_len;
  unsigned int num_slots;
```

(continues on next page)

(continued from previous page)

```
  unsigned int enabled;
  void (*handler)(struct i3c_device *dev, const struct i3c_ibi_payload␣
↪*payload);
};
```

Members

all_ibis_handled used to be informed when no more IBIs are waiting to be processed. Used by `i3c_device_disable_ibi()` to wait for all IBIs to be dequeued

pending_ibis count the number of pending IBIs. Each pending IBI has its work element queued to the controller workqueue

max_payload_len maximum payload length for an IBI coming from this device. this value is specified when calling `i3c_device_request_ibi()` and should not change at run time. All messages IBIs exceeding this limit should be rejected by the master

num_slots number of IBI slots reserved for this device

enabled reflect the IBI status

handler IBI handler specified at `i3c_device_request_ibi()` call time. This handler will be called from the controller workqueue, and as such is allowed to sleep (though it is recommended to process the IBI as fast as possible to not stall processing of other IBIs queued on the same workqueue). New I3C messages can be sent from the IBI handler

Description

The `struct i3c_device_ibi_info` object is allocated when `i3c_device_request_ibi()` is called and attached to a specific device. This object is here to manage IBIs coming from a specific I3C device.

Note that this structure is the generic view of the IBI management infrastructure. I3C master drivers may have their own internal representation which they can associate to the device using controller-private data.

struct **i3c_dev_boardinfo**
I3C device board information

Definition

```
struct i3c_dev_boardinfo {
  struct list_head node;
  u8 init_dyn_addr;
  u8 static_addr;
  u64 pid;
  struct device_node *of_node;
};
```

Members

node used to insert the boardinfo object in the I3C boardinfo list

init_dyn_addr initial dynamic address requested by the FW. We provide no guarantee that the device will end up using this address, but try our best to assign

this specific address to the device

static_addr static address the I3C device listen on before it's been assigned a dynamic address by the master. Will be used during bus initialization to assign it a specific dynamic address before starting DAA (Dynamic Address Assignment)

pid I3C Provisional ID exposed by the device. This is a unique identifier that may be used to attach boardinfo to i3c_dev_desc when the device does not have a static address

of_node optional DT node in case the device has been described in the DT

Description

This structure is used to attach board-level information to an I3C device. Not all I3C devices connected on the bus will have a boardinfo. It's only needed if you want to attach extra resources to a device or assign it a specific dynamic address.

struct **i3c_dev_desc**
I3C device descriptor

Definition

```
struct i3c_dev_desc {
  struct i3c_i2c_dev_desc common;
  struct i3c_device_info info;
  struct mutex ibi_lock;
  struct i3c_device_ibi_info *ibi;
  struct i3c_device *dev;
  const struct i3c_dev_boardinfo *boardinfo;
};
```

Members

common common part of the I3C device descriptor

info I3C device information. Will be automatically filled when you create your device with `i3c_master_add_i3c_dev_locked()`

ibi_lock lock used to protect the `struct_i3c_device->ibi`

ibi IBI info attached to a device. Should be NULL until `i3c_device_request_ibi()` is called

dev pointer to the I3C device object exposed to I3C device drivers. This should never be accessed from I3C master controller drivers. Only core code should manipulate it in when updating the dev <-> desc link or when propagating IBI events to the driver

boardinfo pointer to the boardinfo attached to this I3C device

Description

Internal representation of an I3C device. This object is only used by the core and passed to I3C master controller drivers when they're requested to do some operations on the device. The core maintains the link between the internal I3C dev descriptor and the object exposed to the I3C device drivers (`struct_i3c_device`).

struct **i3c_device**
I3C device object

Definition

```
struct i3c_device {
  struct device dev;
  struct i3c_dev_desc *desc;
  struct i3c_bus *bus;
};
```

Members

dev device object to register the I3C dev to the device model

desc pointer to an i3c device descriptor object. This link is updated every time the I3C device is rediscovered with a different dynamic address assigned

bus I3C bus this device is attached to

Description

I3C device object exposed to I3C device drivers. The takes care of linking this object to the relevant `struct_i3c_dev_desc` one. All I3C devs on the I3C bus are represented, including I3C masters. For each of them, we have an instance of `struct i3c_device`.

enum **i3c_bus_mode**
I3C bus mode

Constants

I3C_BUS_MODE_PURE only I3C devices are connected to the bus. No limitation expected

I3C_BUS_MODE_MIXED_FAST I2C devices with 50ns spike filter are present on the bus. The only impact in this mode is that the high SCL pulse has to stay below 50ns to trick I2C devices when transmitting I3C frames

I3C_BUS_MODE_MIXED_LIMITED I2C devices without 50ns spike filter are present on the bus. However they allow compliance up to the maximum SDR SCL clock frequency.

I3C_BUS_MODE_MIXED_SLOW I2C devices without 50ns spike filter are present on the bus

enum **i3c_addr_slot_status**
I3C address slot status

Constants

I3C_ADDR_SLOT_FREE address is free

I3C_ADDR_SLOT_RSVD address is reserved

I3C_ADDR_SLOT_I2C_DEV address is assigned to an I2C device

I3C_ADDR_SLOT_I3C_DEV address is assigned to an I3C device

I3C_ADDR_SLOT_STATUS_MASK address slot mask

Description

On an I3C bus, addresses are assigned dynamically, and we need to know which addresses are free to use and which ones are already assigned.

Addresses marked as reserved are those reserved by the I3C protocol (broadcast address, ⋯).

struct **i3c_bus**
I3C bus object

Definition

```
struct i3c_bus {
  struct i3c_dev_desc *cur_master;
  int id;
  unsigned long addrslots[((I2C_MAX_ADDR + 1) * 2) / BITS_PER_LONG];
  enum i3c_bus_mode mode;
  struct {
    unsigned long i3c;
    unsigned long i2c;
  } scl_rate;
  struct {
    struct list_head i3c;
    struct list_head i2c;
  } devs;
  struct rw_semaphore lock;
};
```

Members

cur_master I3C master currently driving the bus. Since I3C is multi-master this can change over the time. Will be used to let a master know whether it needs to request bus ownership before sending a frame or not

id bus ID. Assigned by the framework when register the bus

addrslots a bitmap with 2-bits per-slot to encode the address status and ease the DAA (Dynamic Address Assignment) procedure (see `enum i3c_addr_slot_status`)

mode bus mode (see `enum i3c_bus_mode`)

scl_rate SCL signal rate for I3C and I2C mode

scl_rate.i3c maximum rate for the clock signal when doing I3C SDR/priv transfers

scl_rate.i2c maximum rate for the clock signal when doing I2C transfers

devs 2 lists containing all I3C/I2C devices connected to the bus

devs.i3c contains a list of I3C device descriptors representing I3C devices connected on the bus and successfully attached to the I3C master

devs.i2c contains a list of I2C device descriptors representing I2C devices connected on the bus and successfully attached to the I3C master

lock read/write lock on the bus. This is needed to protect against operations that have an impact on the whole bus and the devices connected to it. For example, when asking slaves to drop their dynamic address (RSTDAA CCC), we need to

make sure no one is trying to send I3C frames to these devices. Note that this lock does not protect against concurrency between devices: several drivers can send different I3C/I2C frames through the same master in parallel. This is the responsibility of the master to guarantee that frames are actually sent sequentially and not interlaced

Description

The I3C bus is represented with its own object and not implicitly described by the I3C master to cope with the multi-master functionality, where one bus can be shared amongst several masters, each of them requesting bus ownership when they need to.

struct **i3c_master_controller_ops**
I3C master methods

Definition

```
struct i3c_master_controller_ops {
  int (*bus_init)(struct i3c_master_controller *master);
  void (*bus_cleanup)(struct i3c_master_controller *master);
  int (*attach_i3c_dev)(struct i3c_dev_desc *dev);
  int (*reattach_i3c_dev)(struct i3c_dev_desc *dev, u8 old_dyn_addr);
  void (*detach_i3c_dev)(struct i3c_dev_desc *dev);
  int (*do_daa)(struct i3c_master_controller *master);
  bool (*supports_ccc_cmd)(struct i3c_master_controller *master, const␣
↪struct i3c_ccc_cmd *cmd);
  int (*send_ccc_cmd)(struct i3c_master_controller *master, struct i3c_ccc_
↪cmd *cmd);
  int (*priv_xfers)(struct i3c_dev_desc *dev,struct i3c_priv_xfer *xfers,␣
↪int nxfers);
  int (*attach_i2c_dev)(struct i2c_dev_desc *dev);
  void (*detach_i2c_dev)(struct i2c_dev_desc *dev);
  int (*i2c_xfers)(struct i2c_dev_desc *dev, const struct i2c_msg *xfers,␣
↪int nxfers);
  int (*request_ibi)(struct i3c_dev_desc *dev, const struct i3c_ibi_setup␣
↪*req);
  void (*free_ibi)(struct i3c_dev_desc *dev);
  int (*enable_ibi)(struct i3c_dev_desc *dev);
  int (*disable_ibi)(struct i3c_dev_desc *dev);
  void (*recycle_ibi_slot)(struct i3c_dev_desc *dev, struct i3c_ibi_slot␣
↪*slot);
};
```

Members

bus_init hook responsible for the I3C bus initialization. You should at least call master_set_info() from there and set the bus mode. You can also put controller specific initialization in there. This method is mandatory.

bus_cleanup cleanup everything done in `i3c_master_controller_ops->bus_init()`. This method is optional.

attach_i3c_dev called every time an I3C device is attached to the bus. It can be after a DAA or when a device is statically declared by the FW, in which case it will only have a static address and the dynamic address will be 0. When this function is called, device information have not been retrieved yet. This

is a good place to attach master controller specific data to I3C devices. This method is optional.

reattach_i3c_dev called every time an I3C device has its addressed changed. It can be because the device has been powered down and has lost its address, or it can happen when a device had a static address and has been assigned a dynamic address with SETDASA. This method is optional.

detach_i3c_dev called when an I3C device is detached from the bus. Usually happens when the master device is unregistered. This method is optional.

do_daa do a DAA (Dynamic Address Assignment) procedure. This is procedure should send an ENTDAA CCC command and then add all devices discovered sure the DAA using `i3c_master_add_i3c_dev_locked()`. Add devices added with `i3c_master_add_i3c_dev_locked()` will then be attached or reattached to the controller. This method is mandatory.

supports_ccc_cmd should return true if the CCC command is supported, false otherwise. This method is optional, if not provided the core assumes all CCC commands are supported.

send_ccc_cmd send a CCC command This method is mandatory.

priv_xfers do one or several private I3C SDR transfers This method is mandatory.

attach_i2c_dev called every time an I2C device is attached to the bus. This is a good place to attach master controller specific data to I2C devices. This method is optional.

detach_i2c_dev called when an I2C device is detached from the bus. Usually happens when the master device is unregistered. This method is optional.

i2c_xfers do one or several I2C transfers. Note that, unlike i3c transfers, the core does not guarantee that buffers attached to the transfers are DMA-safe. If drivers want to have DMA-safe buffers, they should use the `i2c_get_dma_safe_msg_buf()` and `i2c_put_dma_safe_msg_buf()` helpers provided by the I2C framework. This method is mandatory.

request_ibi attach an IBI handler to an I3C device. This implies defining an IBI handler and the constraints of the IBI (maximum payload length and number of pre-allocated slots). Some controllers support less IBI-capable devices than regular devices, so this method might return `-EBUSY` if there' s no more space for an extra IBI registration This method is optional.

free_ibi free an IBI previously requested with ->request_ibi(). The IBI should have been disabled with ->disable_irq() prior to that This method is mandatory only if ->request_ibi is not NULL.

enable_ibi enable the IBI. Only valid if ->request_ibi() has been called prior to ->enable_ibi(). The controller should first enable the IBI on the controller end (for example, unmask the hardware IRQ) and then send the ENEC CCC command (with the IBI flag set) to the I3C device. This method is mandatory only if ->request_ibi is not NULL.

disable_ibi disable an IBI. First send the DISEC CCC command with the IBI flag set and then deactivate the hardware IRQ on the controller end. This method is mandatory only if ->request_ibi is not NULL.

recycle_ibi_slot recycle an IBI slot. Called every time an IBI has been processed by its handler. The IBI slot should be put back in the IBI slot pool so that the controller can re-use it for a future IBI This method is mandatory only if ->request_ibi is not NULL.

struct **i3c_master_controller**
I3C master controller object

Definition

```
struct i3c_master_controller {
  struct device dev;
  struct i3c_dev_desc *this;
  struct i2c_adapter i2c;
  const struct i3c_master_controller_ops *ops;
  unsigned int secondary : 1;
  unsigned int init_done : 1;
  struct {
    struct list_head i3c;
    struct list_head i2c;
  } boardinfo;
  struct i3c_bus bus;
  struct workqueue_struct *wq;
};
```

Members

dev device to be registered to the device-model

this an I3C device object representing this master. This device will be added to the list of I3C devs available on the bus

i2c I2C adapter used for backward compatibility. This adapter is registered to the I2C subsystem to be as transparent as possible to existing I2C drivers

ops master operations. See `struct i3c_master_controller_ops`

secondary true if the master is a secondary master

init_done true when the bus initialization is done

boardinfo board-level information attached to devices connected on the bus

boardinfo.i3c list of I3C boardinfo objects

boardinfo.i2c list of I2C boardinfo objects

bus I3C bus exposed by this master

wq workqueue used to execute IBI handlers. Can also be used by master drivers if they need to postpone operations that need to take place in a thread context. Typical examples are Hot Join processing which requires taking the bus lock in maintenance, which in turn, can only be done from a sleep-able context

Description

A `struct i3c_master_controller` has to be registered to the I3C subsystem through `i3c_master_register()`. None of `struct i3c_master_controller` fields should be set manually, just pass appropriate values to `i3c_master_register()`.

i3c_bus_for_each_i2cdev(bus, dev)
iterate over all I2C devices present on the bus

Parameters

bus the I3C bus

dev an I2C device descriptor pointer updated to point to the current slot at each iteration of the loop

Description

Iterate over all I2C devs present on the bus.

i3c_bus_for_each_i3cdev(bus, dev)
iterate over all I3C devices present on the bus

Parameters

bus the I3C bus

dev and I3C device descriptor pointer updated to point to the current slot at each iteration of the loop

Description

Iterate over all I3C devs present on the bus.

void * **i3c_dev_get_master_data**(const struct i3c_dev_desc * dev)
get master private data attached to an I3C device descriptor

Parameters

const struct i3c_dev_desc * dev the I3C device descriptor to get private data from

Return

the private data previously attached with i3c_dev_set_master_data() or NULL if no data has been attached to the device.

void **i3c_dev_set_master_data**(struct i3c_dev_desc * dev, void * data)
attach master private data to an I3C device descriptor

Parameters

struct i3c_dev_desc * dev the I3C device descriptor to attach private data to

void * data private data

Description

This functions allows a master controller to attach per-device private data which can then be retrieved with `i3c_dev_get_master_data()`.

void * **i2c_dev_get_master_data**(const struct i2c_dev_desc * dev)
get master private data attached to an I2C device descriptor

Parameters

const struct i2c_dev_desc * dev the I2C device descriptor to get private data from

Return

the private data previously attached with i2c_dev_set_master_data() or NULL if no data has been attached to the device.

void **i2c_dev_set_master_data**(struct i2c_dev_desc * dev, void * data)
attach master private data to an I2C device descriptor

Parameters

struct i2c_dev_desc * dev the I2C device descriptor to attach private data to

void * data private data

Description

This functions allows a master controller to attach per-device private data which can then be retrieved with i2c_device_get_master_data().

struct i3c_master_controller * **i3c_dev_get_master**(struct i3c_dev_desc * dev)
get master used to communicate with a device

Parameters

struct i3c_dev_desc * dev I3C dev

Return

the master controller driving **dev**

struct i3c_master_controller * **i2c_dev_get_master**(struct i2c_dev_desc * dev)
get master used to communicate with a device

Parameters

struct i2c_dev_desc * dev I2C dev

Return

the master controller driving **dev**

struct i3c_bus * **i3c_master_get_bus**(struct i3c_master_controller * master)
get the bus attached to a master

Parameters

struct i3c_master_controller * master master object

Return

the I3C bus **master** is connected to

CHAPTER

TWENTYEIGHT

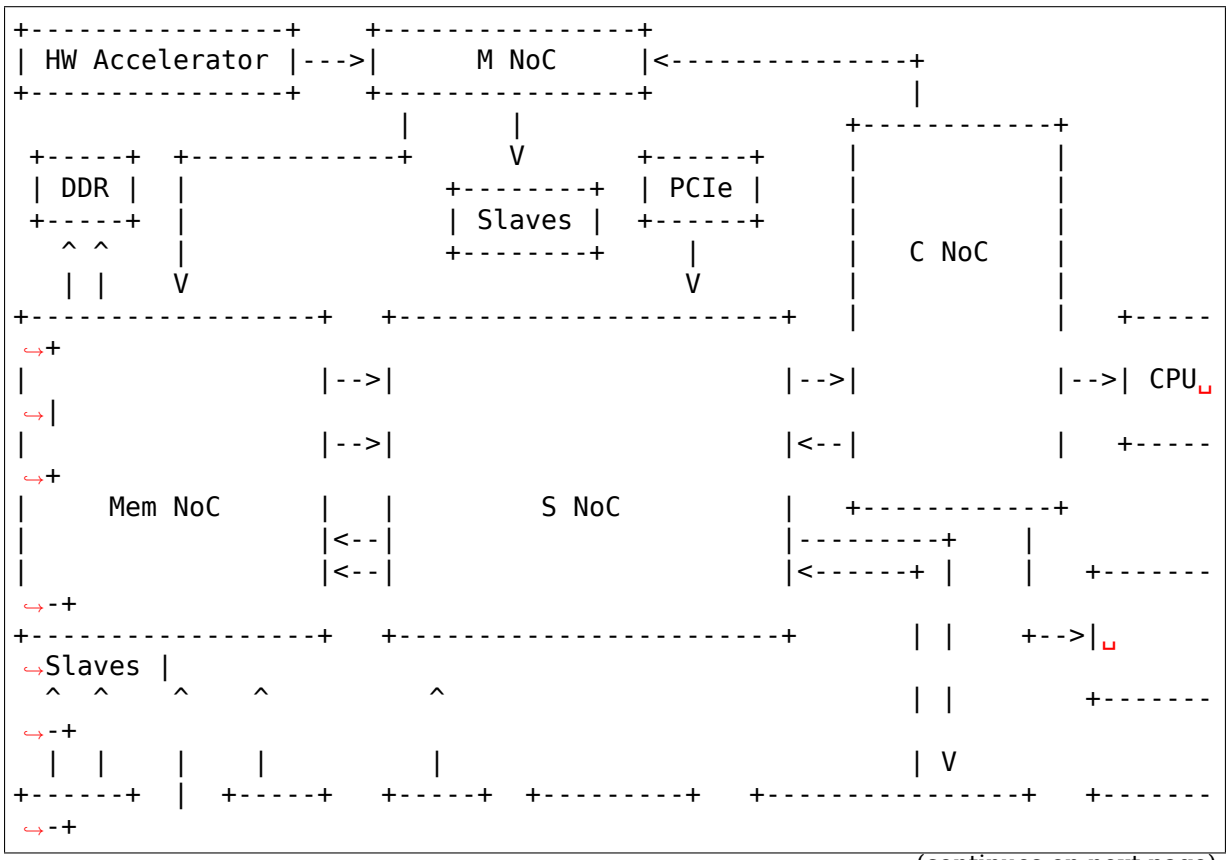
GENERIC SYSTEM INTERCONNECT SUBSYSTEM

28.1 Introduction

This framework is designed to provide a standard kernel interface to control the settings of the interconnects on an SoC. These settings can be throughput, latency and priority between multiple interconnected devices or functional blocks. This can be controlled dynamically in order to save power or provide maximum performance.

The interconnect bus is hardware with configurable parameters, which can be set on a data path according to the requests received from various drivers. An example of interconnect buses are the interconnects between various components or functional blocks in chipsets. There can be multiple interconnects on an SoC that can be multi-tiered.

Below is a simplified diagram of a real-world SoC interconnect bus topology.



(continues on next page)

(continued from previous page)

```
| CPUs |  |  | GPU |   | DSP |  | Masters |-->|       P NoC    |-->|␣
↪Slaves |
+------+  |  +-----+   +-----+  +---------+   +----------------+   +-------
↪-+
          |
      +-------+
      | Modem |
      +-------+
```

28.2 Terminology

Interconnect provider is the software definition of the interconnect hardware. The interconnect providers on the above diagram are M NoC, S NoC, C NoC, P NoC and Mem NoC.

Interconnect node is the software definition of the interconnect hardware port. Each interconnect provider consists of multiple interconnect nodes, which are connected to other SoC components including other interconnect providers. The point on the diagram where the CPUs connect to the memory is called an interconnect node, which belongs to the Mem NoC interconnect provider.

Interconnect endpoints are the first or the last element of the path. Every endpoint is a node, but not every node is an endpoint.

Interconnect path is everything between two endpoints including all the nodes that have to be traversed to reach from a source to destination node. It may include multiple master-slave pairs across several interconnect providers.

Interconnect consumers are the entities which make use of the data paths exposed by the providers. The consumers send requests to providers requesting various throughput, latency and priority. Usually the consumers are device drivers, that send request based on their needs. An example for a consumer is a video decoder that supports various formats and image sizes.

28.3 Interconnect providers

Interconnect provider is an entity that implements methods to initialize and configure interconnect bus hardware. The interconnect provider drivers should be registered with the interconnect provider core.

struct **icc_onecell_data**
driver data for onecell interconnect providers

Definition

```
struct icc_onecell_data {
  unsigned int num_nodes;
  struct icc_node *nodes[];
};
```

Members

num_nodes number of nodes in this device

nodes array of pointers to the nodes in this device

struct **icc_provider**
interconnect provider (controller) entity that might provide multiple interconnect controls

Definition

```
struct icc_provider {
  struct list_head        provider_list;
  struct list_head        nodes;
  int (*set)(struct icc_node *src, struct icc_node *dst);
  int (*aggregate)(struct icc_node *node, u32 tag, u32 avg_bw, u32 peak_bw,
↪ u32 *agg_avg, u32 *agg_peak);
  void (*pre_aggregate)(struct icc_node *node);
  struct icc_node* (*xlate)(struct of_phandle_args *spec, void *data);
  struct device           *dev;
  int users;
  void *data;
};
```

Members

provider_list list of the registered interconnect providers

nodes internal list of the interconnect provider nodes

set pointer to device specific set operation function

aggregate pointer to device specific aggregate operation function

pre_aggregate pointer to device specific function that is called before the aggregation begins (optional)

xlate provider-specific callback for mapping nodes from phandle arguments

dev the device this interconnect provider belongs to

users count of active users

data pointer to private data

struct **icc_node**
entity that is part of the interconnect topology

Definition

```
struct icc_node {
  int id;
  const char              *name;
  struct icc_node         **links;
  size_t num_links;
  struct icc_provider     *provider;
  struct list_head        node_list;
  struct list_head        search_list;
  struct icc_node         *reverse;
  u8 is_traversed:1;
  struct hlist_head       req_list;
  u32 avg_bw;
```

(continues on next page)

(continued from previous page)

```
  u32 peak_bw;
  void *data;
};
```

Members

id platform specific node id

name node name used in debugfs

links a list of targets pointing to where we can go next when traversing

num_links number of links to other interconnect nodes

provider points to the interconnect provider of this node

node_list the list entry in the parent provider's "nodes" list

search_list list used when walking the nodes graph

reverse pointer to previous node when walking the nodes graph

is_traversed flag that is used when walking the nodes graph

req_list a list of QoS constraint requests associated with this node

avg_bw aggregated value of average bandwidth requests from all consumers

peak_bw aggregated value of peak bandwidth requests from all consumers

data pointer to private data

28.4 Interconnect consumers

Interconnect consumers are the clients which use the interconnect APIs to get paths between endpoints and set their bandwidth/latency/QoS requirements for these interconnect paths. These interfaces are not currently documented.

28.5 Interconnect debugfs interfaces

Like several other subsystems interconnect will create some files for debugging and introspection. Files in debugfs are not considered ABI so application software shouldn't rely on format details change between kernel versions.

`/sys/kernel/debug/interconnect/interconnect_summary`:

Show all interconnect nodes in the system with their aggregated bandwidth request. Indented under each node show bandwidth requests from each device.

`/sys/kernel/debug/interconnect/interconnect_graph`:

Show the interconnect graph in the graphviz dot format. It shows all interconnect nodes and links in the system and groups together nodes from the same provider as subgraphs. The format is human-readable and can also be piped through dot to generate diagrams in many graphical formats:

```
$ cat /sys/kernel/debug/interconnect/interconnect_graph | \
        dot -Tsvg > interconnect_graph.svg
```

CHAPTER

TWENTYNINE

DEVICE FREQUENCY SCALING

29.1 Introduction

This framework provides a standard kernel interface for Dynamic Voltage and Frequency Switching on arbitrary devices.

It exposes controls for adjusting frequency through sysfs files which are similar to the cpufreq subsystem.

Devices for which current usage can be measured can have their frequency automatically adjusted by governors.

29.2 API

Device drivers need to initialize a `devfreq_profile` and call the `devfreq_add_device()` function to create a `devfreq` instance.

struct **devfreq_dev_status**
: Data given from devfreq user device to governors. Represents the performance statistics.

Definition

```
struct devfreq_dev_status {
  unsigned long total_time;
  unsigned long busy_time;
  unsigned long current_frequency;
  void *private_data;
};
```

Members

total_time The total time represented by this instance of devfreq_dev_status

busy_time The time that the device was working among the total_time.

current_frequency The operating frequency.

private_data An entry not specified by the devfreq framework. A device and a specific governor may have their own protocol with private_data. However, because this is governor-specific, a governor using this will be only compatible with devices aware of it.

struct **devfreq_dev_profile**
Devfreq's user device profile

Definition

```
struct devfreq_dev_profile {
  unsigned long initial_freq;
  unsigned int polling_ms;
  int (*target)(struct device *dev, unsigned long *freq, u32 flags);
  int (*get_dev_status)(struct device *dev, struct devfreq_dev_status␣
↪*stat);
  int (*get_cur_freq)(struct device *dev, unsigned long *freq);
  void (*exit)(struct device *dev);
  unsigned long *freq_table;
  unsigned int max_state;
};
```

Members

initial_freq The operating frequency when `devfreq_add_device()` is called.

polling_ms The polling interval in ms. 0 disables polling.

target The device should set its operating frequency at freq or lowest-upper-than-freq value. If freq is higher than any operable frequency, set maximum. Before returning, target function should set freq at the current frequency. The "flags" parameter's possible values are explained above with "DEVFREQ_FLAG_*" macros.

get_dev_status The device should provide the current performance status to devfreq. Governors are recommended not to use this directly. Instead, governors are recommended to use devfreq_update_stats() along with devfreq.last_status.

get_cur_freq The device should provide the current frequency at which it is operating.

exit An optional callback that is called when devfreq is removing the devfreq object due to error or from `devfreq_remove_device()` call. If the user has registered devfreq->nb at a notifier-head, this is the time to unregister it.

freq_table Optional list of frequencies to support statistics and freq_table must be generated in ascending order.

max_state The size of freq_table.

struct **devfreq_stats**
Statistics of devfreq device behavior

Definition

```
struct devfreq_stats {
  unsigned int total_trans;
  unsigned int *trans_table;
  u64 *time_in_state;
  u64 last_update;
};
```

Members

total_trans Number of devfreq transitions.

trans_table Statistics of devfreq transitions.

time_in_state Statistics of devfreq states.

last_update The last time stats were updated.

struct **devfreq**
Device devfreq structure

Definition

```
struct devfreq {
  struct list_head node;
  struct mutex lock;
  struct device dev;
  struct devfreq_dev_profile *profile;
  const struct devfreq_governor *governor;
  char governor_name[DEVFREQ_NAME_LEN];
  struct notifier_block nb;
  struct delayed_work work;
  unsigned long previous_freq;
  struct devfreq_dev_status last_status;
  void *data;
  struct dev_pm_qos_request user_min_freq_req;
  struct dev_pm_qos_request user_max_freq_req;
  unsigned long scaling_min_freq;
  unsigned long scaling_max_freq;
  bool stop_polling;
  unsigned long suspend_freq;
  unsigned long resume_freq;
  atomic_t suspend_count;
  struct devfreq_stats stats;
  struct srcu_notifier_head transition_notifier_list;
  struct notifier_block nb_min;
  struct notifier_block nb_max;
};
```

Members

node list node - contains the devices with devfreq that have been registered.

lock a mutex to protect accessing devfreq.

dev device registered by devfreq class. dev.parent is the device using devfreq.

profile device-specific devfreq profile

governor method how to choose frequency based on the usage.

governor_name devfreq governor name for use with this devfreq

nb notifier block used to notify devfreq object that it should reevaluate operable frequencies. Devfreq users may use devfreq.nb to the corresponding register notifier call chain.

work delayed work for load monitoring.

previous_freq previously configured frequency value.

last_status devfreq user device info, performance statistics

data Private data of the governor. The devfreq framework does not touch this.

user_min_freq_req PM QoS minimum frequency request from user (via sysfs)

user_max_freq_req PM QoS maximum frequency request from user (via sysfs)

scaling_min_freq Limit minimum frequency requested by OPP interface

scaling_max_freq Limit maximum frequency requested by OPP interface

stop_polling devfreq polling status of a device.

suspend_freq frequency of a device set during suspend phase.

resume_freq frequency of a device set in resume phase.

suspend_count suspend requests counter for a device.

stats Statistics of devfreq device behavior

transition_notifier_list list head of DEVFREQ_TRANSITION_NOTIFIER notifier

nb_min Notifier block for DEV_PM_QOS_MIN_FREQUENCY

nb_max Notifier block for DEV_PM_QOS_MAX_FREQUENCY

Description

This structure stores the devfreq information for a given device.

Note that when a governor accesses entries in struct devfreq in its functions except for the context of callbacks defined in struct devfreq_governor, the governor should protect its access with the struct mutex lock in struct devfreq. A governor may use this mutex to protect its own private data in `void *data` as well.

int **update_devfreq**(struct devfreq * devfreq)
Reevaluate the device and configure frequency

Parameters

struct devfreq * devfreq the devfreq device

Note

devfreq->lock must be held

struct **devfreq_simple_ondemand_data**
`void *data` fed to struct devfreq and devfreq_add_device

Definition

```
struct devfreq_simple_ondemand_data {
  unsigned int upthreshold;
  unsigned int downdifferential;
};
```

Members

upthreshold If the load is over this value, the frequency jumps. Specify 0 to use the default. Valid value = 0 to 100.

downdifferential If the load is under upthreshold - downdifferential, the governor may consider slowing the frequency down. Specify 0 to use the default. Valid value = 0 to 100. downdifferential < upthreshold must hold.

Description

If the fed devfreq_simple_ondemand_data pointer is NULL to the governor, the governor uses the default values.

struct **devfreq_passive_data**
`void *data` fed to struct devfreq and devfreq_add_device

Definition

```
struct devfreq_passive_data {
  struct devfreq *parent;
  int (*get_target_freq)(struct devfreq *this, unsigned long *freq);
  struct devfreq *this;
  struct notifier_block nb;
};
```

Members

parent the devfreq instance of parent device.

get_target_freq Optional callback, Returns desired operating frequency for the device using passive governor. That is called when passive governor should decide the next frequency by using the new frequency of parent devfreq device using governors except for passive governor. If the devfreq device has the specific method to decide the next frequency, should use this callback.

this the devfreq instance of own device.

nb the notifier block for DEVFREQ_TRANSITION_NOTIFIER list

Description

The devfreq_passive_data have to set the devfreq instance of parent device with governors except for the passive governor. But, don't need to initialize the 'this' and 'nb' field because the devfreq core will handle them.

struct **devfreq_event_dev**
the devfreq-event device

Definition

```
struct devfreq_event_dev {
  struct list_head node;
  struct device dev;
  struct mutex lock;
  u32 enable_count;
  const struct devfreq_event_desc *desc;
};
```

Members

node Contain the devfreq-event device that have been registered.

dev the device registered by devfreq-event class. dev.parent is the device using devfreq-event.

lock a mutex to protect accessing devfreq-event.

enable_count the number of enable function have been called.

desc the description for devfreq-event device.

Description

This structure contains devfreq-event device information.

struct **devfreq_event_data**
the devfreq-event data

Definition

```
struct devfreq_event_data {
  unsigned long load_count;
  unsigned long total_count;
};
```

Members

load_count load count of devfreq-event device for the given period.

total_count total count of devfreq-event device for the given period. each count may represent a clock cycle, a time unit (ns/us/…), or anything the device driver wants. Generally, utilization is load_count / total_count.

Description

This structure contains the data of devfreq-event device for polling period.

struct **devfreq_event_ops**
the operations of devfreq-event device

Definition

```
struct devfreq_event_ops {
  int (*enable)(struct devfreq_event_dev *edev);
  int (*disable)(struct devfreq_event_dev *edev);
  int (*reset)(struct devfreq_event_dev *edev);
  int (*set_event)(struct devfreq_event_dev *edev);
  int (*get_event)(struct devfreq_event_dev *edev, struct devfreq_event_
↪data *edata);
};
```

Members

enable Enable the devfreq-event device.

disable Disable the devfreq-event device.

reset Reset all setting of the devfreq-event device.

set_event Set the specific event type for the devfreq-event device.

get_event Get the result of the devfreq-event devie with specific event type.

Description

This structure contains devfreq-event device operations which can be implemented by devfreq-event device drivers.

struct **devfreq_event_desc**
the descriptor of devfreq-event device

Definition

```
struct devfreq_event_desc {
  const char *name;
  u32 event_type;
  void *driver_data;
  const struct devfreq_event_ops *ops;
};
```

Members

name the name of devfreq-event device.

event_type the type of the event determined and used by driver

driver_data the private data for devfreq-event driver.

ops the operation to control devfreq-event device.

Description

Each devfreq-event device is described with a this structure. This structure contains the various data for devfreq-event device. The event_type describes what is going to be counted in the register. It might choose to count e.g. read requests, write data in bytes, etc. The full supported list of types is present in specyfic header in: include/dt-bindings/pmu/.

int **devfreq_update_status**(struct devfreq * devfreq, unsigned long freq)
Update statistics of devfreq behavior

Parameters

struct devfreq * devfreq the devfreq instance

unsigned long freq the update target frequency

int **update_devfreq**(struct devfreq * devfreq)
Reevaluate the device and configure frequency.

Parameters

struct devfreq * devfreq the devfreq instance.

Note

Lock devfreq->lock before calling update_devfreq This function is exported for governors.

void **devfreq_monitor_start**(struct devfreq * devfreq)
Start load monitoring of devfreq instance

Parameters

struct devfreq * devfreq the devfreq instance.

Description

Helper function for starting devfreq device load monitoring. By default delayed work based monitoring is supported. Function to be called from governor in re-

sponse to DEVFREQ_GOV_START event when device is added to devfreq framework.

void **devfreq_monitor_stop**(struct devfreq * devfreq)
Stop load monitoring of a devfreq instance

Parameters

struct devfreq * devfreq the devfreq instance.

Description

Helper function to stop devfreq device load monitoring. Function to be called from governor in response to DEVFREQ_GOV_STOP event when device is removed from devfreq framework.

void **devfreq_monitor_suspend**(struct devfreq * devfreq)
Suspend load monitoring of a devfreq instance

Parameters

struct devfreq * devfreq the devfreq instance.

Description

Helper function to suspend devfreq device load monitoring. Function to be called from governor in response to DEVFREQ_GOV_SUSPEND event or when polling interval is set to zero.

Note

Though this function is same as `devfreq_monitor_stop()`, intentionally kept separate to provide hooks for collecting transition statistics.

void **devfreq_monitor_resume**(struct devfreq * devfreq)
Resume load monitoring of a devfreq instance

Parameters

struct devfreq * devfreq the devfreq instance.

Description

Helper function to resume devfreq device load monitoring. Function to be called from governor in response to DEVFREQ_GOV_RESUME event or when polling interval is set to non-zero.

void **devfreq_update_interval**(struct devfreq * devfreq, unsigned int * delay)
Update device devfreq monitoring interval

Parameters

struct devfreq * devfreq the devfreq instance.

unsigned int * delay new polling interval to be set.

Description

Helper function to set new load monitoring polling interval. Function to be called from governor in response to DEVFREQ_GOV_UPDATE_INTERVAL event.

struct devfreq * **devfreq_add_device**(struct device * dev, struct devfreq_dev_profile * profile, const char * governor_name, void * data)
Add devfreq feature to the device

Parameters

struct device * dev the device to add devfreq feature.

struct devfreq_dev_profile * profile device-specific profile to run devfreq.

const char * governor_name name of the policy to choose frequency.

void * data private data for the governor. The devfreq framework does not touch this value.

int **devfreq_remove_device**(struct devfreq * devfreq)
Remove devfreq feature from a device.

Parameters

struct devfreq * devfreq the devfreq instance to be removed

Description

The opposite of `devfreq_add_device()`.

struct devfreq * **devm_devfreq_add_device**(struct device * dev, struct devfreq_dev_profile * profile, const char * governor_name, void * data)
Resource-managed `devfreq_add_device()`

Parameters

struct device * dev the device to add devfreq feature.

struct devfreq_dev_profile * profile device-specific profile to run devfreq.

const char * governor_name name of the policy to choose frequency.

void * data private data for the governor. The devfreq framework does not touch this value.

Description

This function manages automatically the memory of devfreq device using device resource management and simplify the free operation for memory of devfreq device.

void **devm_devfreq_remove_device**(struct device * dev, struct devfreq * devfreq)
Resource-managed `devfreq_remove_device()`

Parameters

struct device * dev the device from which to remove devfreq feature.

struct devfreq * devfreq the devfreq instance to be removed

int **devfreq_suspend_device**(struct devfreq * devfreq)
Suspend devfreq of a device.

Parameters

struct devfreq * devfreq the devfreq instance to be suspended

Description

This function is intended to be called by the pm callbacks (e.g., runtime_suspend, suspend) of the device driver that holds the devfreq.

int **devfreq_resume_device**(struct devfreq * devfreq)
Resume devfreq of a device.

Parameters

struct devfreq * devfreq the devfreq instance to be resumed

Description

This function is intended to be called by the pm callbacks (e.g., runtime_resume, resume) of the device driver that holds the devfreq.

int **devfreq_add_governor**(struct devfreq_governor * governor)
Add devfreq governor

Parameters

struct devfreq_governor * governor the devfreq governor to be added

int **devfreq_remove_governor**(struct devfreq_governor * governor)
Remove devfreq feature from a device.

Parameters

struct devfreq_governor * governor the devfreq governor to be removed

struct dev_pm_opp * **devfreq_recommended_opp**(struct device * dev, unsigned long * freq, u32 flags)
Helper function to get proper OPP for the freq value given to target callback.

Parameters

struct device * dev The devfreq user device. (parent of devfreq)

unsigned long * freq The frequency given to target function

u32 flags Flags handed from devfreq framework.

Description

The callers are required to call dev_pm_opp_put() for the returned OPP after use.

int **devfreq_register_opp_notifier**(struct device * dev, struct devfreq * devfreq)
Helper function to get devfreq notified for any changes in the OPP availability changes

Parameters

struct device * dev The devfreq user device. (parent of devfreq)

struct devfreq * devfreq The devfreq object.

int **devfreq_unregister_opp_notifier**(struct device * dev, struct devfreq * devfreq)

Helper function to stop getting devfreq notified for any changes in the OPP availability changes anymore.

Parameters

struct device * dev The devfreq user device. (parent of devfreq)

struct devfreq * devfreq The devfreq object.

Description

At exit() callback of devfreq_dev_profile, this must be included if devfreq_recommended_opp is used.

int **devm_devfreq_register_opp_notifier**(struct device * dev, struct devfreq * devfreq)

Resource-managed `devfreq_register_opp_notifier()`

Parameters

struct device * dev The devfreq user device. (parent of devfreq)

struct devfreq * devfreq The devfreq object.

void **devm_devfreq_unregister_opp_notifier**(struct device * dev, struct devfreq * devfreq)

Resource-managed `devfreq_unregister_opp_notifier()`

Parameters

struct device * dev The devfreq user device. (parent of devfreq)

struct devfreq * devfreq The devfreq object.

int **devfreq_register_notifier**(struct devfreq * devfreq, struct notifier_block * nb, unsigned int list)

Register a driver with devfreq

Parameters

struct devfreq * devfreq The devfreq object.

struct notifier_block * nb The notifier block to register.

unsigned int list DEVFREQ_TRANSITION_NOTIFIER.

int **devm_devfreq_register_notifier**(struct device * dev, struct devfreq * devfreq, struct notifier_block * nb, unsigned int list)

Parameters

struct device * dev The devfreq user device. (parent of devfreq)

struct devfreq * devfreq The devfreq object.

struct notifier_block * nb The notifier block to be unregistered.

unsigned int list DEVFREQ_TRANSITION_NOTIFIER.

Description

- Resource-managed `devfreq_register_notifier()`

void **devm_devfreq_unregister_notifier**(struct device * dev, struct devfreq * devfreq, struct notifier_block * nb, unsigned int list)

Parameters

struct device * dev The devfreq user device. (parent of devfreq)

struct devfreq * devfreq The devfreq object.

struct notifier_block * nb The notifier block to be unregistered.

unsigned int list DEVFREQ_TRANSITION_NOTIFIER.

Description

- Resource-managed devfreq_unregister_notifier()

int **devfreq_event_enable_edev**(struct devfreq_event_dev * edev)
Enable the devfreq-event dev and increase the enable_count of devfreq-event dev.

Parameters

struct devfreq_event_dev * edev the devfreq-event device

Description

Note that this function increase the enable_count and enable the devfreq-event device. The devfreq-event device should be enabled before using it by devfreq device.

int **devfreq_event_disable_edev**(struct devfreq_event_dev * edev)
Disable the devfreq-event dev and decrease the enable_count of the devfreq-event dev.

Parameters

struct devfreq_event_dev * edev the devfreq-event device

Description

Note that this function decrease the enable_count and disable the devfreq-event device. After the devfreq-event device is disabled, devfreq device can't use the devfreq-event device for get/set/reset operations.

bool **devfreq_event_is_enabled**(struct devfreq_event_dev * edev)
Check whether devfreq-event dev is enabled or not.

Parameters

struct devfreq_event_dev * edev the devfreq-event device

Description

Note that this function check whether devfreq-event dev is enabled or not. If return true, the devfreq-event dev is enabeld. If return false, the devfreq-event dev is disabled.

int **devfreq_event_set_event**(struct devfreq_event_dev * edev)
Set event to devfreq-event dev to start.

Parameters

struct devfreq_event_dev * edev the devfreq-event device

Description

Note that this function set the event to the devfreq-event device to start for getting the event data which could be various event type.

int **devfreq_event_get_event**(struct devfreq_event_dev * edev, struct devfreq_event_data * edata)
Get {load|total}_count from devfreq-event dev.

Parameters

struct devfreq_event_dev * edev the devfreq-event device

struct devfreq_event_data * edata the calculated data of devfreq-event device

Description

Note that this function get the calculated event data from devfreq-event dev after stoping the progress of whole sequence of devfreq-event dev.

int **devfreq_event_reset_event**(struct devfreq_event_dev * edev)
Reset all opeations of devfreq-event dev.

Parameters

struct devfreq_event_dev * edev the devfreq-event device

Description

Note that this function stop all operations of devfreq-event dev and reset the current event data to make the devfreq-event device into initial state.

struct devfreq_event_dev * **devfreq_event_get_edev_by_phandle**(struct device * dev, int index)
Get the devfreq-event dev from devicetree.

Parameters

struct device * dev the pointer to the given device

int index the index into list of devfreq-event device

Description

Note that this function return the pointer of devfreq-event device.

int **devfreq_event_get_edev_count**(struct device * dev)
Get the count of devfreq-event dev

Parameters

struct device * dev the pointer to the given device

Description

Note that this function return the count of devfreq-event devices.

struct devfreq_event_dev * **devfreq_event_add_edev**(struct device * dev, struct devfreq_event_desc * desc)
Add new devfreq-event device.

Parameters

struct device * dev the device owning the devfreq-event device being created

struct devfreq_event_desc * desc the devfreq-event device's decriptor which include essential data for devfreq-event device.

Description

Note that this function add new devfreq-event device to devfreq-event class list and register the device of the devfreq-event device.

int **devfreq_event_remove_edev**(struct devfreq_event_dev * edev)
Remove the devfreq-event device registered.

Parameters

struct devfreq_event_dev * edev the devfreq-event device

Description

Note that this function removes the registered devfreq-event device.

struct devfreq_event_dev * **devm_devfreq_event_add_edev**(struct device * dev, struct devfreq_event_desc * desc)
Resource-managed `devfreq_event_add_edev()`

Parameters

struct device * dev the device owning the devfreq-event device being created

struct devfreq_event_desc * desc the devfreq-event device's decriptor which include essential data for devfreq-event device.

Description

Note that this function manages automatically the memory of devfreq-event device using device resource management and simplify the free operation for memory of devfreq-event device.

void **devm_devfreq_event_remove_edev**(struct device * dev, struct devfreq_event_dev * edev)
Resource-managed `devfreq_event_remove_edev()`

Parameters

struct device * dev the device owning the devfreq-event device being created

struct devfreq_event_dev * edev the devfreq-event device

Description

Note that this function manages automatically the memory of devfreq-event device using device resource management.

CHAPTER

THIRTY

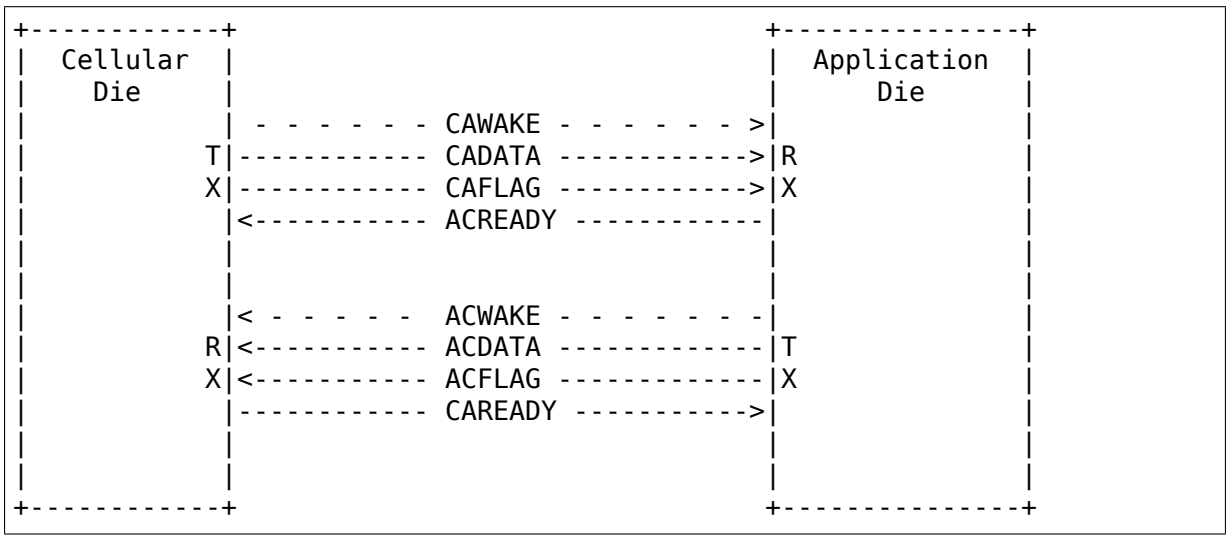
HIGH SPEED SYNCHRONOUS SERIAL INTERFACE (HSI)

30.1 Introduction

High Speed Syncronous Interface (HSI) is a fullduplex, low latency protocol, that is optimized for die-level interconnect between an Application Processor and a Baseband chipset. It has been specified by the MIPI alliance in 2003 and implemented by multiple vendors since then.

The HSI interface supports full duplex communication over multiple channels (typically 8) and is capable of reaching speeds up to 200 Mbit/s.

The serial protocol uses two signals, DATA and FLAG as combined data and clock signals and an additional READY signal for flow control. An additional WAKE signal can be used to wakeup the chips from standby modes. The signals are commonly prefixed by AC for signals going from the application die to the cellular die and CA for signals going the other way around.



30.2 HSI Subsystem in Linux

In the Linux kernel the hsi subsystem is supposed to be used for HSI devices. The hsi subsystem contains drivers for hsi controllers including support for multi-port controllers and provides a generic API for using the HSI ports.

It also contains HSI client drivers, which make use of the generic API to implement a protocol used on the HSI interface. These client drivers can use an arbitrary number of channels.

30.3 hsi-char Device

Each port automatically registers a generic client driver called hsi_char, which provides a charecter device for userspace representing the HSI port. It can be used to communicate via HSI from userspace. Userspace may configure the hsi_char device using the following ioctl commands:

HSC_RESET flush the HSI port

HSC_SET_PM enable or disable the client.

HSC_SEND_BREAK send break

HSC_SET_RX set RX configuration

HSC_GET_RX get RX configuration

HSC_SET_TX set TX configuration

HSC_GET_TX get TX configuration

30.4 The kernel HSI API

struct **hsi_channel**
    channel resource used by the hsi clients

Definition

```
struct hsi_channel {
  unsigned int    id;
  const char      *name;
};
```

Members

id Channel number

name Channel name

struct **hsi_config**
    Configuration for RX/TX HSI modules

Definition

```
struct hsi_config {
  unsigned int            mode;
  struct hsi_channel      *channels;
  unsigned int            num_channels;
  unsigned int            num_hw_channels;
  unsigned int            speed;
  union {
    unsigned int    flow;
    unsigned int    arb_mode;
  };
};
```

Members

mode Bit transmission mode (STREAM or FRAME)

channels Channel resources used by the client

num_channels Number of channel resources

num_hw_channels Number of channels the transceiver is configured for [1..16]

speed Max bit transmission speed (Kbit/s)

{unnamed_union} anonymous

flow RX flow type (SYNCHRONIZED or PIPELINE)

arb_mode Arbitration mode for TX frame (Round robin, priority)

struct **hsi_board_info**
HSI client board info

Definition

```
struct hsi_board_info {
  const char              *name;
  unsigned int            hsi_id;
  unsigned int            port;
  struct hsi_config       tx_cfg;
  struct hsi_config       rx_cfg;
  void *platform_data;
  struct dev_archdata     *archdata;
};
```

Members

name Name for the HSI device

hsi_id HSI controller id where the client sits

port Port number in the controller where the client sits

tx_cfg HSI TX configuration

rx_cfg HSI RX configuration

platform_data Platform related data

archdata Architecture-dependent device data

struct **hsi_client**
HSI client attached to an HSI port

Definition

```
struct hsi_client {
  struct device           device;
  struct hsi_config       tx_cfg;
  struct hsi_config       rx_cfg;
};
```

Members

device Driver model representation of the device

tx_cfg HSI TX configuration

rx_cfg HSI RX configuration

struct **hsi_client_driver**
Driver associated to an HSI client

Definition

```
struct hsi_client_driver {
  struct device_driver    driver;
};
```

Members

driver Driver model representation of the driver

struct **hsi_msg**
HSI message descriptor

Definition

```
struct hsi_msg {
  struct list_head        link;
  struct hsi_client       *cl;
  struct sg_table         sgt;
  void *context;
  void (*complete)(struct hsi_msg *msg);
  void (*destructor)(struct hsi_msg *msg);
  int status;
  unsigned int            actual_len;
  unsigned int            channel;
  unsigned int            ttype:1;
  unsigned int            break_frame:1;
};
```

Members

link Free to use by the current descriptor owner

cl HSI device client that issues the transfer

sgt Head of the scatterlist array

context Client context data associated to the transfer

complete Transfer completion callback

destructor Destructor to free resources when flushing

status Status of the transfer when completed

actual_len Actual length of data transferred on completion

channel Channel were to TX/RX the message

ttype Transfer type (TX if set, RX otherwise)

break_frame if true HSI will send/receive a break frame. Data buffers are ignored in the request.

struct **hsi_port**
HSI port device

Definition

```
struct hsi_port {
  struct device                 device;
  struct hsi_config             tx_cfg;
  struct hsi_config             rx_cfg;
  unsigned int                  num;
  unsigned int                  shared:1;
  int claimed;
  struct mutex                  lock;
  int (*async)(struct hsi_msg *msg);
  int (*setup)(struct hsi_client *cl);
  int (*flush)(struct hsi_client *cl);
  int (*start_tx)(struct hsi_client *cl);
  int (*stop_tx)(struct hsi_client *cl);
  int (*release)(struct hsi_client *cl);
  struct blocking_notifier_head   n_head;
};
```

Members

device Driver model representation of the device

tx_cfg Current TX path configuration

rx_cfg Current RX path configuration

num Port number

shared Set when port can be shared by different clients

claimed Reference count of clients which claimed the port

lock Serialize port claim

async Asynchronous transfer callback

setup Callback to set the HSI client configuration

flush Callback to clean the HW state and destroy all pending transfers

start_tx Callback to inform that a client wants to TX data

stop_tx Callback to inform that a client no longer wishes to TX data

release Callback to inform that a client no longer uses the port

n_head Notifier chain for signaling port events to the clients.

struct **hsi_controller**
HSI controller device

Definition

```
struct hsi_controller {
  struct device           device;
  struct module           *owner;
  unsigned int            id;
  unsigned int            num_ports;
  struct hsi_port         **port;
};
```

Members

device Driver model representation of the device

owner Pointer to the module owning the controller

id HSI controller ID

num_ports Number of ports in the HSI controller

port Array of HSI ports

unsigned int **hsi_id**(struct hsi_client * cl)
Get HSI controller ID associated to a client

Parameters

struct hsi_client * cl Pointer to a HSI client

Description

Return the controller id where the client is attached to

unsigned int **hsi_port_id**(struct hsi_client * cl)
Gets the port number a client is attached to

Parameters

struct hsi_client * cl Pointer to HSI client

Description

Return the port number associated to the client

int **hsi_setup**(struct hsi_client * cl)
Configure the client' s port

Parameters

struct hsi_client * cl Pointer to the HSI client

Description

When sharing ports, clients should either relay on a single client setup or have the same setup for all of them.

Return -errno on failure, 0 on success

int **hsi_flush**(struct hsi_client * cl)
Flush all pending transactions on the client's port

Parameters

struct hsi_client * cl Pointer to the HSI client

Description

This function will destroy all pending hsi_msg in the port and reset the HW port so it is ready to receive and transmit from a clean state.

Return -errno on failure, 0 on success

int **hsi_async_read**(struct hsi_client * cl, struct hsi_msg * msg)
Submit a read transfer

Parameters

struct hsi_client * cl Pointer to the HSI client

struct hsi_msg * msg HSI message descriptor of the transfer

Description

Return -errno on failure, 0 on success

int **hsi_async_write**(struct hsi_client * cl, struct hsi_msg * msg)
Submit a write transfer

Parameters

struct hsi_client * cl Pointer to the HSI client

struct hsi_msg * msg HSI message descriptor of the transfer

Description

Return -errno on failure, 0 on success

int **hsi_start_tx**(struct hsi_client * cl)
Signal the port that the client wants to start a TX

Parameters

struct hsi_client * cl Pointer to the HSI client

Description

Return -errno on failure, 0 on success

int **hsi_stop_tx**(struct hsi_client * cl)
Signal the port that the client no longer wants to transmit

Parameters

struct hsi_client * cl Pointer to the HSI client

Description

Return -errno on failure, 0 on success

void **hsi_port_unregister_clients**(struct hsi_port * port)
Unregister an HSI port

Parameters

struct hsi_port * port The HSI port to unregister

void **hsi_unregister_controller**(struct hsi_controller * hsi)
Unregister an HSI controller

Parameters

struct hsi_controller * hsi The HSI controller to register

int **hsi_register_controller**(struct hsi_controller * hsi)
Register an HSI controller and its ports

Parameters

struct hsi_controller * hsi The HSI controller to register

Description

Returns -errno on failure, 0 on success.

int **hsi_register_client_driver**(struct hsi_client_driver * drv)
Register an HSI client to the HSI bus

Parameters

struct hsi_client_driver * drv HSI client driver to register

Description

Returns -errno on failure, 0 on success.

void **hsi_put_controller**(struct hsi_controller * hsi)
Free an HSI controller

Parameters

struct hsi_controller * hsi Pointer to the HSI controller to freed

Description

HSI controller drivers should only use this function if they need to free their allocated hsi_controller structures before a successful call to hsi_register_controller. Other use is not allowed.

struct hsi_controller * **hsi_alloc_controller**(unsigned int n_ports, gfp_t flags)
Allocate an HSI controller and its ports

Parameters

unsigned int n_ports Number of ports on the HSI controller

gfp_t flags Kernel allocation flags

Description

Return NULL on failure or a pointer to an hsi_controller on success.

void **hsi_free_msg**(struct hsi_msg * msg)
Free an HSI message

Parameters

struct hsi_msg * msg Pointer to the HSI message

Description

Client is responsible to free the buffers pointed by the scatterlists.

struct hsi_msg * **hsi_alloc_msg**(unsigned int nents, gfp_t flags)
Allocate an HSI message

Parameters

unsigned int nents Number of memory entries

gfp_t flags Kernel allocation flags

Description

nents can be 0. This mainly makes sense for read transfer. In that case, HSI drivers will call the complete callback when there is data to be read without consuming it.

Return NULL on failure or a pointer to an hsi_msg on success.

int **hsi_async**(struct hsi_client * cl, struct hsi_msg * msg)
Submit an HSI transfer to the controller

Parameters

struct hsi_client * cl HSI client sending the transfer

struct hsi_msg * msg The HSI transfer passed to controller

Description

The HSI message must have the channel, ttype, complete and destructor fields set beforehand. If nents > 0 then the client has to initialize also the scatterlists to point to the buffers to write to or read from.

HSI controllers relay on pre-allocated buffers from their clients and they do not allocate buffers on their own.

Once the HSI message transfer finishes, the HSI controller calls the complete callback with the status and actual_len fields of the HSI message updated. The complete callback can be called before returning from hsi_async.

Returns -errno on failure or 0 on success

int **hsi_claim_port**(struct hsi_client * cl, unsigned int share)
Claim the HSI client's port

Parameters

struct hsi_client * cl HSI client that wants to claim its port

unsigned int share Flag to indicate if the client wants to share the port or not.

Description

Returns -errno on failure, 0 on success.

void **hsi_release_port**(struct hsi_client * cl)
Release the HSI client's port

Parameters

struct hsi_client * cl HSI client which previously claimed its port

int **hsi_register_port_event**(struct hsi_client * cl, void (*handler)(struct hsi_client *, unsigned long))
Register a client to receive port events

Parameters

struct hsi_client * cl HSI client that wants to receive port events

void (*)(struct hsi_client *, unsigned long) handler Event handler callback

Description

Clients should register a callback to be able to receive events from the ports. Registration should happen after claiming the port. The handler can be called in interrupt context.

Returns -errno on error, or 0 on success.

int **hsi_unregister_port_event**(struct hsi_client * cl)
Stop receiving port events for a client

Parameters

struct hsi_client * cl HSI client that wants to stop receiving port events

Description

Clients should call this function before releasing their associated port.

Returns -errno on error, or 0 on success.

int **hsi_event**(struct hsi_port * port, unsigned long event)
Notifies clients about port events

Parameters

struct hsi_port * port Port where the event occurred

unsigned long event The event type

Description

Clients should not be concerned about wake line behavior. However, due to a race condition in HSI HW protocol, clients need to be notified about wake line changes, so they can implement a workaround for it.

Events: HSI_EVENT_START_RX - Incoming wake line high HSI_EVENT_STOP_RX - Incoming wake line down

Returns -errno on error, or 0 on success.

int **hsi_get_channel_id_by_name**(struct hsi_client * cl, char * name)
acquire channel id by channel name

Parameters

struct hsi_client * cl HSI client, which uses the channel

char * name name the channel is known under

Description

Clients can call this function to get the hsi channel ids similar to requesting IRQs or GPIOs by name. This function assumes the same channel configuration is used for RX and TX.

Returns -errno on error or channel id on success.

CHAPTER

THIRTYONE

ERROR DETECTION AND CORRECTION (EDAC) DEVICES

31.1 Main Concepts used at the EDAC subsystem

There are several things to be aware of that aren't at all obvious, like sockets, *socket sets, banks, rows, chip-select rows, channels, etc…

These are some of the many terms that are thrown about that don't always mean what people think they mean (Inconceivable!). In the interest of creating a common ground for discussion, terms and their definitions will be established.

- Memory devices

The individual DRAM chips on a memory stick. These devices commonly output 4 and 8 bits each (x4, x8). Grouping several of these in parallel provides the number of bits that the memory controller expects: typically 72 bits, in order to provide 64 bits + 8 bits of ECC data.

- Memory Stick

A printed circuit board that aggregates multiple memory devices in parallel. In general, this is the Field Replaceable Unit (FRU) which gets replaced, in the case of excessive errors. Most often it is also called DIMM (Dual Inline Memory Module).

- Memory Socket

A physical connector on the motherboard that accepts a single memory stick. Also called as "slot" on several datasheets.

- Channel

A memory controller channel, responsible to communicate with a group of DIMMs. Each channel has its own independent control (command) and data bus, and can be used independently or grouped with other channels.

- Branch

It is typically the highest hierarchy on a Fully-Buffered DIMM memory controller. Typically, it contains two channels. Two channels at the same branch can be used in single mode or in lockstep mode. When lockstep is enabled, the cacheline is doubled, but it generally brings some performance penalty. Also, it is generally not possible to point to just one memory stick when an error occurs, as the error correction code is calculated using two DIMMs instead of one. Due to that, it is capable of correcting more errors than on single mode.

- Single-channel

The data accessed by the memory controller is contained into one dimm only. E. g. if the data is 64 bits-wide, the data flows to the CPU using one 64 bits parallel access. Typically used with SDR, DDR, DDR2 and DDR3 memories. FB-DIMM and RAMBUS use a different concept for channel, so this concept doesn't apply there.

- Double-channel

The data size accessed by the memory controller is interlaced into two dimms, accessed at the same time. E. g. if the DIMM is 64 bits-wide (72 bits with ECC), the data flows to the CPU using a 128 bits parallel access.

- Chip-select row

This is the name of the DRAM signal used to select the DRAM ranks to be accessed. Common chip-select rows for single channel are 64 bits, for dual channel 128 bits. It may not be visible by the memory controller, as some DIMM types have a memory buffer that can hide direct access to it from the Memory Controller.

- Single-Ranked stick

A Single-ranked stick has 1 chip-select row of memory. Motherboards commonly drive two chip-select pins to a memory stick. A single-ranked stick, will occupy only one of those rows. The other will be unused.

- Double-Ranked stick

A double-ranked stick has two chip-select rows which access different sets of memory devices. The two rows cannot be accessed concurrently.

- Double-sided stick

DEPRECATED TERM, see Double-Ranked stick.

A double-sided stick has two chip-select rows which access different sets of memory devices. The two rows cannot be accessed concurrently. "Double-sided" is irrespective of the memory devices being mounted on both sides of the memory stick.

- Socket set

All of the memory sticks that are required for a single memory access or all of the memory sticks spanned by a chip-select row. A single socket set has two chip-select rows and if double-sided sticks are used these will occupy those chip-select rows.

- Bank

This term is avoided because it is unclear when needing to distinguish between chip-select rows and socket sets.

31.2 Memory Controllers

Most of the EDAC core is focused on doing Memory Controller error detection. The `edac_mc_alloc()`. It uses internally the struct `mem_ctl_info` to describe the memory controllers, with is an opaque struct for the EDAC drivers. Only the EDAC core is allowed to touch it.

enum **dev_type**
describe the type of memory DRAM chips used at the stick

Constants

DEV_UNKNOWN Can't be determined, or MC doesn't support detect it

DEV_X1 1 bit for data

DEV_X2 2 bits for data

DEV_X4 4 bits for data

DEV_X8 8 bits for data

DEV_X16 16 bits for data

DEV_X32 32 bits for data

DEV_X64 64 bits for data

Description

Typical values are x4 and x8.

enum **hw_event_mc_err_type**
type of the detected error

Constants

HW_EVENT_ERR_CORRECTED Corrected Error - Indicates that an ECC corrected error was detected

HW_EVENT_ERR_UNCORRECTED Uncorrected Error - Indicates an error that can't be corrected by ECC, but it is not fatal (maybe it is on an unused memory area, or the memory controller could recover from it for example, by re-trying the operation).

HW_EVENT_ERR_DEFERRED Deferred Error - Indicates an uncorrectable error whose handling is not urgent. This could be due to hardware data poisoning where the system can continue operation until the poisoned data is consumed. Preemptive measures may also be taken, e.g. offlining pages, etc.

HW_EVENT_ERR_FATAL Fatal Error - Uncorrected error that could not be recovered.

HW_EVENT_ERR_INFO Informational - The CPER spec defines a forth type of error: informational logs.

enum **mem_type**
memory types. For a more detailed reference, please see http://en.wikipedia.org/wiki/DRAM

Constants

MEM_EMPTY Empty csrow

MEM_RESERVED Reserved csrow type

MEM_UNKNOWN Unknown csrow type

MEM_FPM FPM - Fast Page Mode, used on systems up to 1995.

MEM_EDO EDO - Extended data out, used on systems up to 1998.

MEM_BEDO BEDO - Burst Extended data out, an EDO variant.

MEM_SDR SDR - Single data rate SDRAM http://en.wikipedia.org/wiki/Synchronous_dynamic_random-access_memory They use 3 pins for chip select: Pins 0 and 2 are for rank 0; pins 1 and 3 are for rank 1, if the memory is dual-rank.

MEM_RDR Registered SDR SDRAM

MEM_DDR Double data rate SDRAM http://en.wikipedia.org/wiki/DDR_SDRAM

MEM_RDDR Registered Double data rate SDRAM This is a variant of the DDR memories. A registered memory has a buffer inside it, hiding part of the memory details to the memory controller.

MEM_RMBS Rambus DRAM, used on a few Pentium III/IV controllers.

MEM_DDR2 DDR2 RAM, as described at JEDEC JESD79-2F. Those memories are labeled as "PC2-" instead of "PC" to differentiate from DDR.

MEM_FB_DDR2 Fully-Buffered DDR2, as described at JEDEC Std No. 205 and JESD206. Those memories are accessed per DIMM slot, and not by a chip select signal.

MEM_RDDR2 Registered DDR2 RAM This is a variant of the DDR2 memories.

MEM_XDR Rambus XDR It is an evolution of the original RAMBUS memories, created to compete with DDR2. Weren't used on any x86 arch, but cell_edac PPC memory controller uses it.

MEM_DDR3 DDR3 RAM

MEM_RDDR3 Registered DDR3 RAM This is a variant of the DDR3 memories.

MEM_LRDDR3 Load-Reduced DDR3 memory.

MEM_DDR4 Unbuffered DDR4 RAM

MEM_RDDR4 Registered DDR4 RAM This is a variant of the DDR4 memories.

MEM_LRDDR4 Load-Reduced DDR4 memory.

MEM_NVDIMM Non-volatile RAM

enum **edac_type**
type - Error Detection and Correction capabilities and mode

Constants

EDAC_UNKNOWN Unknown if ECC is available

EDAC_NONE Doesn't support ECC

EDAC_RESERVED Reserved ECC type

EDAC_PARITY Detects parity errors

EDAC_EC Error Checking - no correction

EDAC_SECDED Single bit error correction, Double detection

EDAC_S2ECD2ED Chipkill x2 devices - do these exist?

EDAC_S4ECD4ED Chipkill x4 devices

EDAC_S8ECD8ED Chipkill x8 devices

EDAC_S16ECD16ED Chipkill x16 devices

enum **scrub_type**
scrubbing capabilities

Constants

SCRUB_UNKNOWN Unknown if scrubber is available

SCRUB_NONE No scrubber

SCRUB_SW_PROG SW progressive (sequential) scrubbing

SCRUB_SW_SRC Software scrub only errors

SCRUB_SW_PROG_SRC Progressive software scrub from an error

SCRUB_SW_TUNABLE Software scrub frequency is tunable

SCRUB_HW_PROG HW progressive (sequential) scrubbing

SCRUB_HW_SRC Hardware scrub only errors

SCRUB_HW_PROG_SRC Progressive hardware scrub from an error

SCRUB_HW_TUNABLE Hardware scrub frequency is tunable

enum **edac_mc_layer_type**
memory controller hierarchy layer

Constants

EDAC_MC_LAYER_BRANCH memory layer is named “branch”

EDAC_MC_LAYER_CHANNEL memory layer is named “channel”

EDAC_MC_LAYER_SLOT memory layer is named “slot”

EDAC_MC_LAYER_CHIP_SELECT memory layer is named “chip select”

EDAC_MC_LAYER_ALL_MEM memory layout is unknown. All memory is mapped as a single memory area. This is used when retrieving errors from a firmware driven driver.

Description

This enum is used by the drivers to tell edac_mc_sysfs what name should be used when describing a memory stick location.

struct **edac_mc_layer**
describes the memory controller hierarchy

Definition

```
struct edac_mc_layer {
  enum edac_mc_layer_type type;
  unsigned size;
  bool is_virt_csrow;
};
```

Members

type layer type

size number of components per layer. For example, if the channel layer has two channels, size = 2

is_virt_csrow This layer is part of the "csrow" when old API compatibility mode is enabled. Otherwise, it is a channel

struct **rank_info**
contains the information for one DIMM rank

Definition

```
struct rank_info {
  int chan_idx;
  struct csrow_info *csrow;
  struct dimm_info *dimm;
  u32 ce_count;
};
```

Members

chan_idx channel number where the rank is (typically, 0 or 1)

csrow A pointer to the chip select row structure (the parent structure). The location of the rank is given by the (csrow->csrow_idx, chan_idx) vector.

dimm A pointer to the DIMM structure, where the DIMM label information is stored.

ce_count number of correctable errors for this rank

Description

FIXME: Currently, the EDAC core model will assume one DIMM per rank.
This is a bad assumption, but it makes this patch easier. Later patches in this series will fix this issue.

struct **edac_raw_error_desc**
Raw error report structure

Definition

```
struct edac_raw_error_desc {
  char location[LOCATION_SIZE];
  char label[(EDAC_MC_LABEL_LEN + 1 + sizeof(OTHER_LABEL)) * EDAC_MAX_
↪LABELS];
  long grain;
  u16 error_count;
  enum hw_event_mc_err_type type;
  int top_layer;
```

(continues on next page)

(continued from previous page)

```
  int mid_layer;
  int low_layer;
  unsigned long page_frame_number;
  unsigned long offset_in_page;
  unsigned long syndrome;
  const char *msg;
  const char *other_detail;
};
```

Members

location location of the error

label label of the affected DIMM(s)

grain minimum granularity for an error report, in bytes

error_count number of errors of the same type

type severity of the error (CE/UE/Fatal)

top_layer top layer of the error (layer[0])

mid_layer middle layer of the error (layer[1])

low_layer low layer of the error (layer[2])

page_frame_number page where the error happened

offset_in_page page offset

syndrome syndrome of the error (or 0 if unknown or if the syndrome is not applicable)

msg error message

other_detail other driver-specific detail about the error

struct dimm_info * **edac_get_dimm_by_index**(struct mem_ctl_info * mci, int index)

Get DIMM info at **index** from a memory controller

Parameters

struct mem_ctl_info * mci MC descriptor struct mem_ctl_info

int index index in the memory controller's DIMM array

Description

Returns a struct dimm_info * or NULL on failure.

struct dimm_info * **edac_get_dimm**(struct mem_ctl_info * mci, int layer0, int layer1, int layer2)

Get DIMM info from a memory controller given by [layer0,layer1,layer2] position

Parameters

struct mem_ctl_info * mci MC descriptor struct mem_ctl_info

int layer0 layer0 position

int layer1 layer1 position. Unused if n_layers < 2

int layer2 layer2 position. Unused if n_layers < 3

Description

For 1 layer, this function returns “dimms[layer0]”;

For 2 layers, this function is similar to allocating a two-dimensional array and returning “dimms[layer0][layer1]”;

For 3 layers, this function is similar to allocating a tri-dimensional array and returning “dimms[layer0][layer1][layer2]”;

struct mem_ctl_info * **edac_mc_alloc**(unsigned int mc_num, unsigned int n_layers, struct edac_mc_layer * layers, unsigned int sz_pvt)

Allocate and partially fill a struct `mem_ctl_info`.

Parameters

unsigned int mc_num Memory controller number

unsigned int n_layers Number of MC hierarchy layers

struct edac_mc_layer * layers Describes each layer as seen by the Memory Controller

unsigned int sz_pvt size of private storage needed

Description

Everything is kmalloc’ed as one big chunk - more efficient. Only can be used if all structures have the same lifetime - otherwise you have to allocate and initialize your own structures.

Use `edac_mc_free()` to free mc structures allocated by this function.

Note: drivers handle multi-rank memories in different ways: in some drivers, one multi-rank memory stick is mapped as one entry, while, in others, a single multi-rank memory stick would be mapped into several entries. Currently, this function will allocate multiple struct dimm_info on such scenarios, as grouping the multiple ranks require drivers change.

Return

On success, return a pointer to struct mem_ctl_info pointer; `NULL` otherwise

const char * **edac_get_owner**(void)

Return the owner’s mod_name of EDAC MC

Parameters

void no arguments

Return

Pointer to mod_name string when EDAC MC is owned. NULL otherwise.

void **edac_mc_free**(struct mem_ctl_info * mci)
Frees a previously allocated **mci** structure

Parameters

struct mem_ctl_info * mci pointer to a struct mem_ctl_info structure

bool **edac_has_mcs**(void)
Check if any MCs have been allocated.

Parameters

void no arguments

Return

True if MC instances have been registered successfully. False otherwise.

struct mem_ctl_info * **edac_mc_find**(int idx)
Search for a mem_ctl_info structure whose index is **idx**.

Parameters

int idx index to be seek

Description

If found, return a pointer to the structure. Else return NULL.

struct mem_ctl_info * **find_mci_by_dev**(struct device * dev)
Scan list of controllers looking for the one that manages the **dev** device.

Parameters

struct device * dev pointer to a struct device related with the MCI

Return

on success, returns a pointer to struct `mem_ctl_info`; `NULL` otherwise.

struct mem_ctl_info * **edac_mc_del_mc**(struct device * dev)
Remove sysfs entries for mci structure associated with **dev** and remove mci structure from global list.

Parameters

struct device * dev Pointer to struct `device` representing mci structure to remove.

Return

pointer to removed mci structure, or `NULL` if device not found.

int **edac_mc_find_csrow_by_page**(struct mem_ctl_info * mci, unsigned long page)
Ancillary routine to identify what csrow contains a memory page.

Parameters

struct mem_ctl_info * mci pointer to a struct mem_ctl_info structure

unsigned long page memory page to find

Return

on success, returns the csrow. -1 if not found.

void **edac_raw_mc_handle_error**(struct edac_raw_error_desc * e)
Reports a memory event to userspace without doing anything to discover the error location.

Parameters

struct edac_raw_error_desc * e error description

Description

This raw function is used internally by `edac_mc_handle_error()`. It should only be called directly when the hardware error come directly from BIOS, like in the case of APEI GHES driver.

void **edac_mc_handle_error**(const enum hw_event_mc_err_type type, struct mem_ctl_info * mci, const u16 error_count, const unsigned long page_frame_number, const unsigned long offset_in_page, const unsigned long syndrome, const int top_layer, const int mid_layer, const int low_layer, const char * msg, const char * other_detail)
Reports a memory event to userspace.

Parameters

const enum hw_event_mc_err_type type severity of the error (CE/UE/Fatal)

struct mem_ctl_info * mci a struct mem_ctl_info pointer

const u16 error_count Number of errors of the same type

const unsigned long page_frame_number mem page where the error occurred

const unsigned long offset_in_page offset of the error inside the page

const unsigned long syndrome ECC syndrome

const int top_layer Memory layer[0] position

const int mid_layer Memory layer[1] position

const int low_layer Memory layer[2] position

const char * msg Message meaningful to the end users that explains the event

const char * other_detail Technical details about the event that may help hardware manufacturers and EDAC developers to analyse the event

31.3 PCI Controllers

The EDAC subsystem provides a mechanism to handle PCI controllers by calling the `edac_pci_alloc_ctl_info()`. It will use the struct `edac_pci_ctl_info` to describe the PCI controllers.

struct edac_pci_ctl_info * **edac_pci_alloc_ctl_info**(unsigned int sz_pvt, const char * edac_pci_name)

Parameters

unsigned int sz_pvt size of the private info at struct `edac_pci_ctl_info`

const char * edac_pci_name name of the PCI device

Description

The alloc() function for the 'edac_pci' control info structure.

The chip driver will allocate one of these for each edac_pci it is going to control/register with the EDAC CORE.

Return

a pointer to struct `edac_pci_ctl_info` on success; `NULL` otherwise.

void **edac_pci_free_ctl_info**(struct edac_pci_ctl_info * pci)

Parameters

struct edac_pci_ctl_info * pci pointer to struct `edac_pci_ctl_info`

Description

Last action on the pci control structure.

Calls the remove sysfs information, which will unregister this control struct's kobj. When that kobj's ref count goes to zero, its release function will be call and then kfree() the memory.

int **edac_pci_alloc_index**(void)

Parameters

void no arguments

Return

allocated index number

int **edac_pci_add_device**(struct edac_pci_ctl_info * pci, int edac_idx)

Parameters

struct edac_pci_ctl_info * pci pointer to the edac_device structure to be added to the list

int edac_idx A unique numeric identifier to be assigned to the 'edac_pci' structure.

Description

edac_pci global list and create sysfs entries associated with edac_pci structure.

Return

0 on Success, or an error code on failure

struct edac_pci_ctl_info * **edac_pci_del_device**(struct device * dev)

Parameters

struct device * dev Pointer to 'struct device' representing edac_pci structure to remove

Description

Remove sysfs entries for specified edac_pci structure and then remove edac_pci structure from global list

Return

Pointer to removed edac_pci structure, or NULL if device not found

struct edac_pci_ctl_info * **edac_pci_create_generic_ctl**(struct device * dev, const char * mod_name)

Parameters

struct device * dev pointer to struct device;

const char * mod_name name of the PCI device

Description

A generic constructor for a PCI parity polling device Some systems have more than one domain of PCI busses. For systems with one domain, then this API will provide for a generic poller.

This routine calls the edac_pci_alloc_ctl_info() for the generic device, with default values

Return

Pointer to struct edac_pci_ctl_info on success, NULL on failure.

void **edac_pci_release_generic_ctl**(struct edac_pci_ctl_info * pci)

Parameters

struct edac_pci_ctl_info * pci pointer to struct edac_pci_ctl_info

Description

The release function of a generic EDAC PCI polling device

int **edac_pci_create_sysfs**(struct edac_pci_ctl_info * pci)

Parameters

struct edac_pci_ctl_info * pci pointer to struct edac_pci_ctl_info

Description

Create the controls/attributes for the specified EDAC PCI device

void **edac_pci_remove_sysfs**(struct edac_pci_ctl_info * pci)

Parameters

struct edac_pci_ctl_info * pci pointer to struct edac_pci_ctl_info

Description

remove the controls and attributes for this EDAC PCI device

31.4 EDAC Blocks

The EDAC subsystem also provides a generic mechanism to report errors on other parts of the hardware via `edac_device_alloc_ctl_info()` function.

The structures `edac_dev_sysfs_block_attribute`, `edac_device_block`, `edac_device_instance` and `edac_device_ctl_info` provide a generic or abstract 'edac_device' representation at sysfs.

This set of structures and the code that implements the APIs for the same, provide for registering EDAC type devices which are NOT standard memory or PCI, like:

- CPU caches (L1 and L2)
- DMA engines
- Core CPU switches
- Fabric switch units
- PCIe interface controllers
- other EDAC/ECC type devices that can be monitored for errors, etc.

It allows for a 2 level set of hierarchy.

For example, a cache could be composed of L1, L2 and L3 levels of cache. Each CPU core would have its own L1 cache, while sharing L2 and maybe L3 caches. On such case, those can be represented via the following sysfs nodes:

```
/sys/devices/system/edac/..

pci/            <existing pci directory (if available)>
mc/             <existing memory device directory>
cpu/cpu0/..     <L1 and L2 block directory>
        /L1-cache/ce_count
                 /ue_count
        /L2-cache/ce_count
                 /ue_count
cpu/cpu1/..     <L1 and L2 block directory>
        /L1-cache/ce_count
                 /ue_count
        /L2-cache/ce_count
                 /ue_count
...

the L1 and L2 directories would be "edac_device_block's"
```

int **edac_device_add_device**(struct edac_device_ctl_info * edac_dev)

Parameters

struct edac_device_ctl_info * edac_dev pointer to edac_device structure to be added to the list 'edac_device' structure.

Description

edac_device global list and create sysfs entries associated with edac_device structure.

Return

0 on Success, or an error code on failure

struct edac_device_ctl_info * **edac_device_del_device**(struct device * dev)

Parameters

struct device * dev Pointer to struct `device` representing the edac device structure to remove.

Description

Remove sysfs entries for specified edac_device structure and then remove edac_device structure from global list

Return

Pointer to removed edac_device structure, or `NULL` if device not found.

void **edac_device_handle_ce_count**(struct edac_device_ctl_info * edac_dev, unsigned int count, int inst_nr, int block_nr, const char * msg)

Parameters

struct edac_device_ctl_info * edac_dev pointer to struct `edac_device_ctl_info`

unsigned int count Number of errors to log.

int inst_nr number of the instance where the CE error happened

int block_nr number of the block where the CE error happened

const char * msg message to be printed

void **edac_device_handle_ue_count**(struct edac_device_ctl_info * edac_dev, unsigned int count, int inst_nr, int block_nr, const char * msg)

Parameters

struct edac_device_ctl_info * edac_dev pointer to struct `edac_device_ctl_info`

unsigned int count Number of errors to log.

int inst_nr number of the instance where the CE error happened

int block_nr number of the block where the CE error happened

const char * msg message to be printed

void **edac_device_handle_ce**(struct edac_device_ctl_info * edac_dev, int inst_nr, int block_nr, const char * msg)

Parameters

struct edac_device_ctl_info * edac_dev pointer to struct edac_device_ctl_info

int inst_nr number of the instance where the CE error happened

int block_nr number of the block where the CE error happened

const char * msg message to be printed

void **edac_device_handle_ue**(struct edac_device_ctl_info * edac_dev, int inst_nr, int block_nr, const char * msg)

Parameters

struct edac_device_ctl_info * edac_dev pointer to struct edac_device_ctl_info

int inst_nr number of the instance where the UE error happened

int block_nr number of the block where the UE error happened

const char * msg message to be printed

int **edac_device_alloc_index**(void)

Parameters

void no arguments

Return

allocated index number

CHAPTER

THIRTYTWO

SCSI INTERFACES GUIDE

Author James Bottomley

Author Rob Landley

32.1 Introduction

32.1.1 Protocol vs bus

Once upon a time, the Small Computer Systems Interface defined both a parallel I/O bus and a data protocol to connect a wide variety of peripherals (disk drives, tape drives, modems, printers, scanners, optical drives, test equipment, and medical devices) to a host computer.

Although the old parallel (fast/wide/ultra) SCSI bus has largely fallen out of use, the SCSI command set is more widely used than ever to communicate with devices over a number of different busses.

The SCSI protocol is a big-endian peer-to-peer packet based protocol. SCSI commands are 6, 10, 12, or 16 bytes long, often followed by an associated data payload.

SCSI commands can be transported over just about any kind of bus, and are the default protocol for storage devices attached to USB, SATA, SAS, Fibre Channel, FireWire, and ATAPI devices. SCSI packets are also commonly exchanged over Infiniband, I2O, TCP/IP (iSCSI), even Parallel ports.

32.1.2 Design of the Linux SCSI subsystem

The SCSI subsystem uses a three layer design, with upper, mid, and low layers. Every operation involving the SCSI subsystem (such as reading a sector from a disk) uses one driver at each of the 3 levels: one upper layer driver, one lower layer driver, and the SCSI midlayer.

The SCSI upper layer provides the interface between userspace and the kernel, in the form of block and char device nodes for I/O and ioctl(). The SCSI lower layer contains drivers for specific hardware devices.

In between is the SCSI mid-layer, analogous to a network routing layer such as the IPv4 stack. The SCSI mid-layer routes a packet based data protocol between the upper layer’ s /dev nodes and the corresponding devices in the lower layer.

It manages command queues, provides error handling and power management functions, and responds to ioctl() requests.

32.2 SCSI upper layer

The upper layer supports the user-kernel interface by providing device nodes.

32.2.1 sd (SCSI Disk)

sd (sd_mod.o)

32.2.2 sr (SCSI CD-ROM)

sr (sr_mod.o)

32.2.3 st (SCSI Tape)

st (st.o)

32.2.4 sg (SCSI Generic)

sg (sg.o)

32.2.5 ch (SCSI Media Changer)

ch (ch.c)

32.3 SCSI mid layer

32.3.1 SCSI midlayer implementation

include/scsi/scsi_device.h

struct **scsi_vpd**
SCSI Vital Product Data

Definition

```
struct scsi_vpd {
  struct rcu_head rcu;
  int len;
  unsigned char   data[];
};
```

Members

rcu For kfree_rcu().

len Length in bytes of **data**.

data VPD data as defined in various T10 SCSI standard documents.

shost_for_each_device(sdev, shost)
iterate over all devices of a host

Parameters

sdev the `struct scsi_device` to use as a cursor

shost the `struct scsi_host` to iterate over

Description

Iterator that returns each device attached to **shost**. This loop takes a reference on each device and releases it at the end. If you break out of the loop, you must call scsi_device_put(sdev).

__shost_for_each_device(sdev, shost)
iterate over all devices of a host (UNLOCKED)

Parameters

sdev the `struct scsi_device` to use as a cursor

shost the `struct scsi_host` to iterate over

Description

Iterator that returns each device attached to **shost**. It does _not_ take a reference on the scsi_device, so the whole loop must be protected by shost->host_lock.

Note

The only reason to use this is because you need to access the device list in interrupt context. Otherwise you really want to use shost_for_each_device instead.

int **scsi_device_supports_vpd**(struct scsi_device * sdev)
test if a device supports VPD pages

Parameters

struct scsi_device * sdev the `struct scsi_device` to test

Description

If the 'try_vpd_pages' flag is set it takes precedence. Otherwise we will assume VPD pages are supported if the SCSI level is at least SPC-3 and 'skip_vpd_pages' is not set.

drivers/scsi/scsi.c

Main file for the SCSI midlayer.

int **scsi_change_queue_depth**(struct scsi_device * sdev, int depth)
change a device's queue depth

Parameters

struct scsi_device * sdev SCSI Device in question

int depth number of commands allowed to be queued to the driver

Description

Sets the device queue depth and returns the new value.

int **scsi_track_queue_full**(struct scsi_device * sdev, int depth)
track QUEUE_FULL events to adjust queue depth

Parameters

struct scsi_device * sdev SCSI Device in question

int depth Current number of outstanding SCSI commands on this device, not counting the one returned as QUEUE_FULL.

Description

This function will track successive QUEUE_FULL events on a specific SCSI device to determine if and when there is a need to adjust the queue depth on the device.

Lock Status: None held on entry

Return

0 - No change needed, >0 - Adjust queue depth to this new depth,

-1 - Drop back to untagged operation using host->cmd_per_lun as the untagged command depth

Notes

Low level drivers may call this at any time and we will do "The Right Thing." We are interrupt context safe.

int **scsi_get_vpd_page**(struct scsi_device * sdev, u8 page, unsigned char * buf, int buf_len)
Get Vital Product Data from a SCSI device

Parameters

struct scsi_device * sdev The device to ask

u8 page Which Vital Product Data to return

unsigned char * buf where to store the VPD

int buf_len number of bytes in the VPD buffer area

Description

SCSI devices may optionally supply Vital Product Data. Each 'page' of VPD is defined in the appropriate SCSI document (eg SPC, SBC). If the device supports this VPD page, this routine returns a pointer to a buffer containing the data from that page. The caller is responsible for calling kfree() on this pointer when it is no longer needed. If we cannot retrieve the VPD page this routine returns `NULL`.

int **scsi_report_opcode**(struct scsi_device * sdev, unsigned char * buffer, unsigned int len, unsigned char opcode)
Find out if a given command opcode is supported

Parameters

struct scsi_device * sdev scsi device to query

unsigned char * buffer scratch buffer (must be at least 20 bytes long)

unsigned int len length of buffer

unsigned char opcode opcode for command to look up

Description

Uses the REPORT SUPPORTED OPERATION CODES to look up the given opcode. Returns -EINVAL if RSOC fails, 0 if the command opcode is unsupported and 1 if the device claims to support the command.

int **scsi_device_get**(struct scsi_device * sdev)
get an additional reference to a scsi_device

Parameters

struct scsi_device * sdev device to get a reference to

Description

Gets a reference to the scsi_device and increments the use count of the underlying LLDD module. You must hold host_lock of the parent Scsi_Host or already have a reference when calling this.

This will fail if a device is deleted or cancelled, or when the LLD module is in the process of being unloaded.

void **scsi_device_put**(struct scsi_device * sdev)
release a reference to a scsi_device

Parameters

struct scsi_device * sdev device to release a reference on.

Description

Release a reference to the scsi_device and decrements the use count of the underlying LLDD module. The device is freed once the last user vanishes.

void **starget_for_each_device**(struct scsi_target * starget, void * data,
void (*fn)(struct scsi_device *, void *))
helper to walk all devices of a target

Parameters

struct scsi_target * starget target whose devices we want to iterate over.

void * data Opaque passed to each function call.

void (*)(struct scsi_device *, void *) fn Function to call on each device

Description

This traverses over each device of **starget**. The devices have a reference that must be released by scsi_host_put when breaking out of the loop.

void **__starget_for_each_device**(struct scsi_target * starget, void * data,
void (*fn)(struct scsi_device *, void *))
helper to walk all devices of a target (UNLOCKED)

Parameters

struct scsi_target * starget target whose devices we want to iterate over.

void * data parameter for callback **fn()**

void (*)(struct scsi_device *, void *) fn callback function that is invoked for each device

Description

This traverses over each device of **starget**. It does _not_ take a reference on the scsi_device, so the whole loop must be protected by shost->host_lock.

Note

The only reason why drivers would want to use this is because they need to access the device list in irq context. Otherwise you really want to use starget_for_each_device instead.

struct scsi_device * **__scsi_device_lookup_by_target**(struct scsi_target * starget, u64 lun)

find a device given the target (UNLOCKED)

Parameters

struct scsi_target * starget SCSI target pointer

u64 lun SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified **lun** for a given **starget**. The returned scsi_device does not have an additional reference. You must hold the host's host_lock over this call and any access to the returned scsi_device. A scsi_device in state SDEV_DEL is skipped.

Note

The only reason why drivers should use this is because they need to access the device list in irq context. Otherwise you really want to use scsi_device_lookup_by_target instead.

struct scsi_device * **scsi_device_lookup_by_target**(struct scsi_target * starget, u64 lun)

find a device given the target

Parameters

struct scsi_target * starget SCSI target pointer

u64 lun SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified **lun** for a given **starget**. The returned scsi_device has an additional reference that needs to be released with scsi_device_put once you're done with it.

struct scsi_device * **__scsi_device_lookup**(struct Scsi_Host * shost, uint channel, uint id, u64 lun)

find a device given the host (UNLOCKED)

Parameters

struct Scsi_Host * shost SCSI host pointer

uint channel SCSI channel (zero if only one channel)

uint id SCSI target number (physical unit number)

u64 lun SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified **channel**, **id**, **lun** for a given host. The returned scsi_device does not have an additional reference. You must hold the host's host_lock over this call and any access to the returned scsi_device.

Note

The only reason why drivers would want to use this is because they need to access the device list in irq context. Otherwise you really want to use scsi_device_lookup instead.

struct scsi_device * **scsi_device_lookup**(struct Scsi_Host * shost, uint channel, uint id, u64 lun)
find a device given the host

Parameters

struct Scsi_Host * shost SCSI host pointer

uint channel SCSI channel (zero if only one channel)

uint id SCSI target number (physical unit number)

u64 lun SCSI Logical Unit Number

Description

Looks up the scsi_device with the specified **channel**, **id**, **lun** for a given host. The returned scsi_device has an additional reference that needs to be released with scsi_device_put once you're done with it.

drivers/scsi/scsicam.c

SCSI Common Access Method support functions, for use with HDIO_GETGEO, etc.

unsigned char * **scsi_bios_ptable**(struct block_device * dev)
Read PC partition table out of first sector of device.

Parameters

struct block_device * dev from this device

Description

Reads the first sector from the device and returns 0x42 bytes starting at offset `0x1be`.

Return

partition table in kmalloc(GFP_KERNEL) memory, or NULL on error.

bool **scsi_partsize**(struct block_device * bdev, sector_t capacity, int geom)
Parse cylinders/heads/sectors from PC partition table

Parameters

struct block_device * bdev block device to parse

sector_t capacity size of the disk in sectors

int geom output in form of [hds, cylinders, sectors]

Description

Determine the BIOS mapping/geometry used to create the partition table, storing the results in **geom**.

Return

`false` on failure, `true` on success.

int **scsicam_bios_param**(struct block_device * bdev, sector_t capacity, int * ip)
Determine geometry of a disk in cylinders/heads/sectors.

Parameters

struct block_device * bdev which device

sector_t capacity size of the disk in sectors

int * ip return value: ip[0]=heads, ip[1]=sectors, ip[2]=cylinders

Description

determine the BIOS mapping/geometry used for a drive in a SCSI-CAM system, storing the results in ip as required by the HDIO_GETGEO ioctl().

Return

-1 on failure, 0 on success.

drivers/scsi/scsi_error.c

Common SCSI error/timeout handling routines.

void **scsi_schedule_eh**(struct Scsi_Host * shost)
schedule EH for SCSI host

Parameters

struct Scsi_Host * shost SCSI host to invoke error handling on.

Description

Schedule SCSI EH without scmd.

int **scsi_block_when_processing_errors**(struct scsi_device * sdev)
Prevent cmds from being queued.

Parameters

struct scsi_device * sdev Device on which we are performing recovery.

Description

We block until the host is out of error recovery, and then check to see whether the host or the device is offline.

Return value: 0 when dev was taken offline by error recovery. 1 OK to proceed.

int **scsi_check_sense**(struct scsi_cmnd * scmd)
Examine scsi cmd sense

Parameters

struct scsi_cmnd * scmd Cmd to have sense checked.

Description

Return value: SUCCESS or FAILED or NEEDS_RETRY or ADD_TO_MLQUEUE

Notes

When a deferred error is detected the current command has not been executed and needs retrying.

void **scsi_eh_prep_cmnd**(struct scsi_cmnd * scmd, struct scsi_eh_save * ses, unsigned char * cmnd, int cmnd_size, unsigned sense_bytes)
Save a scsi command info as part of error recovery

Parameters

struct scsi_cmnd * scmd SCSI command structure to hijack

struct scsi_eh_save * ses structure to save restore information

unsigned char * cmnd CDB to send. Can be NULL if no new cmnd is needed

int cmnd_size size in bytes of **cmnd** (must be <= BLK_MAX_CDB)

unsigned sense_bytes size of sense data to copy. or 0 (if != 0 **cmnd** is ignored)

Description

This function is used to save a scsi command information before re-execution as part of the error recovery process. If **sense_bytes** is 0 the command sent must be one that does not transfer any data. If **sense_bytes** != 0 **cmnd** is ignored and this functions sets up a REQUEST_SENSE command and cmnd buffers to read **sense_bytes** into **scmd->sense_buffer**.

void **scsi_eh_restore_cmnd**(struct scsi_cmnd * scmd, struct scsi_eh_save * ses)
Restore a scsi command info as part of error recovery

Parameters

struct scsi_cmnd * scmd SCSI command structure to restore

struct scsi_eh_save * ses saved information from a coresponding call to scsi_eh_prep_cmnd

Description

Undo any damage done by above `scsi_eh_prep_cmnd()`.

void **scsi_eh_finish_cmd**(struct scsi_cmnd * scmd, struct list_head * done_q)
Handle a cmd that eh is finished with.

Parameters

struct scsi_cmnd * scmd Original SCSI cmd that eh has finished.

struct list_head * done_q Queue for processed commands.

Notes

We don't want to use the normal command completion while we are are still handling errors - it may cause other commands to be queued, and that would disturb what we are doing. Thus we really want to keep a list of pending commands for final completion, and once we are ready to leave error handling we handle completion for real.

int **scsi_eh_get_sense**(struct list_head * work_q, struct list_head * done_q)
Get device sense data.

Parameters

struct list_head * work_q Queue of commands to process.

struct list_head * done_q Queue of processed commands.

Description

See if we need to request sense information. if so, then get it now, so we have a better idea of what to do.

Notes

This has the unfortunate side effect that if a shost adapter does not automatically request sense information, we end up shutting it down before we request it.

All drivers should request sense information internally these days, so for now all I have to say is tough noogies if you end up in here.

XXX: Long term this code should go away, but that needs an audit of all LLDDs first.

void **scsi_eh_ready_devs**(struct Scsi_Host * shost, struct list_head * work_q, struct list_head * done_q)
check device ready state and recover if not.

Parameters

struct Scsi_Host * shost host to be recovered.

struct list_head * work_q `list_head` for pending commands.

struct list_head * done_q `list_head` for processed commands.

void **scsi_eh_flush_done_q**(struct list_head * done_q)
finish processed commands or retry them.

Parameters

struct list_head * done_q list_head of processed commands.

bool **scsi_get_sense_info_fld**(const u8 * sense_buffer, int sb_len, u64 * info_out)
get information field from sense data (either fixed or descriptor format)

Parameters

const u8 * sense_buffer byte array of sense data

int sb_len number of valid bytes in sense_buffer

u64 * info_out pointer to 64 integer where 8 or 4 byte information field will be placed if found.

Description

Return value: true if information field found, false if not found.

drivers/scsi/scsi_devinfo.c

Manage scsi_dev_info_list, which tracks blacklisted and whitelisted devices.

int **scsi_dev_info_list_add**(int compatible, char * vendor, char * model, char * strflags, blist_flags_t flags)

add one dev_info list entry.

Parameters

int compatible if true, null terminate short strings. Otherwise space pad.

char * vendor vendor string

char * model model (product) string

char * strflags integer string

blist_flags_t flags if strflags NULL, use this flag value

Description

Create and add one dev_info entry for **vendor**, **model**, **strflags** or **flag**. If **compatible**, add to the tail of the list, do not space pad, and set devinfo->compatible. The scsi_static_device_list entries are added with **compatible** 1 and **clfags** NULL.

Return

0 OK, -error on failure.

struct scsi_dev_info_list * **scsi_dev_info_list_find**(const char * vendor, const char * model, enum scsi_devinfo_key key)

find a matching dev_info list entry.

Parameters

const char * vendor full vendor string

const char * model full model (product) string

enum scsi_devinfo_key key specify list to use

Description

Finds the first dev_info entry matching **vendor**, **model** in list specified by **key**.

Return

pointer to matching entry, or ERR_PTR on failure.

int **scsi_dev_info_list_add_str**(char * dev_list)
parse dev_list and add to the scsi_dev_info_list.

Parameters

char * dev_list string of device flags to add

Description

Parse dev_list, and add entries to the scsi_dev_info_list. dev_list is of the form "vendor:product:flag,vendor:product:flag". dev_list is modified via strsep. Can be called for command line addition, for proc or mabye a sysfs interface.

Return

0 if OK, -error on failure.

blist_flags_t **scsi_get_device_flags**(struct scsi_device * sdev, const unsigned char * vendor, const unsigned char * model)
get device specific flags from the dynamic device list.

Parameters

struct scsi_device * sdev scsi_device to get flags for

const unsigned char * vendor vendor name

const unsigned char * model model name

Description

Search the global scsi_dev_info_list (specified by list zero) for an entry matching **vendor** and **model**, if found, return the matching flags value, else return the host or global default settings. Called during scan time.

void **scsi_exit_devinfo**(void)
remove /proc/scsi/device_info & the scsi_dev_info_list

Parameters

void no arguments

int **scsi_init_devinfo**(void)
set up the dynamic device list.

Parameters

void no arguments

Description

Add command line entries from scsi_dev_flags, then add scsi_static_device_list entries to the scsi device info list.

drivers/scsi/scsi_ioctl.c

Handle ioctl() calls for SCSI devices.

int **scsi_ioctl**(struct scsi_device * sdev, int cmd, void __user * arg)
Dispatch ioctl to scsi device

Parameters

struct scsi_device * sdev scsi device receiving ioctl

int cmd which ioctl is it

void __user * arg data associated with ioctl

Description

The `scsi_ioctl()` function differs from most ioctls in that it does not take a major/minor number as the dev field. Rather, it takes a pointer to a `struct scsi_device`.

drivers/scsi/scsi_lib.c

SCSI queuing library.

int **__scsi_execute**(struct scsi_device * sdev, const unsigned char * cmd, int data_direction, void * buffer, unsigned bufflen, unsigned char * sense, struct scsi_sense_hdr * sshdr, int timeout, int retries, u64 flags, req_flags_t rq_flags, int * resid)
insert request and wait for the result

Parameters

struct scsi_device * sdev scsi device

const unsigned char * cmd scsi command

int data_direction data direction

void * buffer data buffer

unsigned bufflen len of buffer

unsigned char * sense optional sense buffer

struct scsi_sense_hdr * sshdr optional decoded sense header

int timeout request timeout in seconds

int retries number of times to retry request

u64 flags flags for ->cmd_flags

req_flags_t rq_flags flags for ->rq_flags

int * resid optional residual length

Description

Returns the scsi_cmnd result field if a command was executed, or a negative Linux error code if we didn’t get that far.

blk_status_t **scsi_init_io**(struct scsi_cmnd * cmd)
SCSI I/O initialization function.

Parameters

struct scsi_cmnd * cmd command descriptor we wish to initialize

Return

- BLK_STS_OK - on success
- BLK_STS_RESOURCE - if the failure is retryable
- BLK_STS_IOERR - if the failure is fatal

struct scsi_device * **scsi_device_from_queue**(struct request_queue * q)
return sdev associated with a request_queue

Parameters

struct request_queue * q The request queue to return the sdev from

Description

Return the sdev associated with a request queue or NULL if the request_queue does not reference a SCSI device.

void **scsi_block_requests**(struct Scsi_Host * shost)
Utility function used by low-level drivers to prevent further commands from being queued to the device.

Parameters

struct Scsi_Host * shost host in question

Description

There is no timer nor any other means by which the requests get unblocked other than the low-level driver calling `scsi_unblock_requests()`.

void **scsi_unblock_requests**(struct Scsi_Host * shost)
Utility function used by low-level drivers to allow further commands to be queued to the device.

Parameters

struct Scsi_Host * shost host in question

Description

There is no timer nor any other means by which the requests get unblocked other than the low-level driver calling `scsi_unblock_requests()`. This is done as an API function so that changes to the internals of the scsi mid-layer won't require wholesale changes to drivers that use this feature.

int **scsi_mode_select**(struct scsi_device * sdev, int pf, int sp, int modepage, unsigned char * buffer, int len, int timeout, int retries, struct scsi_mode_data * data, struct scsi_sense_hdr * sshdr)
issue a mode select

Parameters

struct scsi_device * sdev SCSI device to be queried

int pf Page format bit (1 == standard, 0 == vendor specific)

int sp Save page bit (0 == don't save, 1 == save)

int modepage mode page being requested

unsigned char * buffer request buffer (may not be smaller than eight bytes)

int len length of request buffer.

int timeout command timeout

int retries number of retries before failing

struct scsi_mode_data * data returns a structure abstracting the mode header data

struct scsi_sense_hdr * sshdr place to put sense data (or NULL if no sense to be collected). must be SCSI_SENSE_BUFFERSIZE big.

Returns zero if successful; negative error number or scsi status on error

int **scsi_mode_sense**(struct scsi_device * sdev, int dbd, int modepage, unsigned char * buffer, int len, int timeout, int retries, struct scsi_mode_data * data, struct scsi_sense_hdr * sshdr)

issue a mode sense, falling back from 10 to six bytes if necessary.

Parameters

struct scsi_device * sdev SCSI device to be queried

int dbd set if mode sense will allow block descriptors to be returned

int modepage mode page being requested

unsigned char * buffer request buffer (may not be smaller than eight bytes)

int len length of request buffer.

int timeout command timeout

int retries number of retries before failing

struct scsi_mode_data * data returns a structure abstracting the mode header data

struct scsi_sense_hdr * sshdr place to put sense data (or NULL if no sense to be collected). must be SCSI_SENSE_BUFFERSIZE big.

Returns zero if unsuccessful, or the header offset (either 4 or 8 depending on whether a six or ten byte command was issued) if successful.

int **scsi_test_unit_ready**(struct scsi_device * sdev, int timeout, int retries, struct scsi_sense_hdr * sshdr)

test if unit is ready

Parameters

struct scsi_device * sdev scsi device to change the state of.

int timeout command timeout

int retries number of retries before failing

struct scsi_sense_hdr * sshdr outpout pointer for decoded sense information.

Returns zero if unsuccessful or an error if TUR failed. For removable media, UNIT_ATTENTION sets ->changed flag.

int **scsi_device_set_state**(struct scsi_device * sdev, enum scsi_device_state state)
Take the given device through the device state model.

Parameters

struct scsi_device * sdev scsi device to change the state of.

enum scsi_device_state state state to change to.

Returns zero if successful or an error if the requested transition is illegal.

void **sdev_evt_send**(struct scsi_device * sdev, struct scsi_event * evt)
send asserted event to uevent thread

Parameters

struct scsi_device * sdev scsi_device event occurred on

struct scsi_event * evt event to send

Assert scsi device event asynchronously.

struct scsi_event * **sdev_evt_alloc**(enum scsi_device_event evt_type, gfp_t gfpflags)
allocate a new scsi event

Parameters

enum scsi_device_event evt_type type of event to allocate

gfp_t gfpflags GFP flags for allocation

Allocates and returns a new scsi_event.

void **sdev_evt_send_simple**(struct scsi_device * sdev, enum scsi_device_event evt_type, gfp_t gfpflags)
send asserted event to uevent thread

Parameters

struct scsi_device * sdev scsi_device event occurred on

enum scsi_device_event evt_type type of event to send

gfp_t gfpflags GFP flags for allocation

Assert scsi device event asynchronously, given an event type.

int **scsi_device_quiesce**(struct scsi_device * sdev)
Block user issued commands.

Parameters

struct scsi_device * sdev scsi device to quiesce.

This works by trying to transition to the SDEV_QUIESCE state (which must be a legal transition). When the device is in this state, only special requests will be accepted, all others will be deferred. Since special requests may also be requeued requests, a successful return doesn't guarantee the device will be totally quiescent.

Must be called with user context, may sleep.

Returns zero if unsuccessful or an error if not.

void **scsi_device_resume**(struct scsi_device * sdev)
Restart user issued commands to a quiesced device.

Parameters

struct scsi_device * sdev scsi device to resume.

Moves the device from quiesced back to running and restarts the queues.

Must be called with user context, may sleep.

int **scsi_internal_device_block_nowait**(struct scsi_device * sdev)
try to transition to the SDEV_BLOCK state

Parameters

struct scsi_device * sdev device to block

Description

Pause SCSI command processing on the specified device. Does not sleep.

Returns zero if successful or a negative error code upon failure.

Notes

This routine transitions the device to the SDEV_BLOCK state (which must be a legal transition). When the device is in this state, command processing is paused until the device leaves the SDEV_BLOCK state. See also `scsi_internal_device_unblock_nowait()`.

int **scsi_internal_device_unblock_nowait**(struct scsi_device * sdev, enum scsi_device_state new_state)
resume a device after a block request

Parameters

struct scsi_device * sdev device to resume

enum scsi_device_state new_state state to set the device to after unblocking

Description

Restart the device queue for a previously suspended SCSI device. Does not sleep.

Returns zero if successful or a negative error code upon failure.

Notes

This routine transitions the device to the SDEV_RUNNING state or to one of the offline states (which must be a legal transition) allowing the midlayer to goose the queue for this device.

void * **scsi_kmap_atomic_sg**(struct scatterlist * sgl, int sg_count, size_t * offset, size_t * len)
find and atomically map an sg-elemnt

Parameters

struct scatterlist * sgl scatter-gather list

int sg_count number of segments in sg

size_t * offset offset in bytes into sg, on return offset into the mapped area

size_t * len bytes to map, on return number of bytes mapped

Description

Returns virtual address of the start of the mapped page

void **scsi_kunmap_atomic_sg**(void * virt)
atomically unmap a virtual address, previously mapped with scsi_kmap_atomic_sg

Parameters

void * virt virtual address to be unmapped

int **scsi_vpd_lun_id**(struct scsi_device * sdev, char * id, size_t id_len)
return a unique device identification

Parameters

struct scsi_device * sdev SCSI device

char * id buffer for the identification

size_t id_len length of the buffer

Description

Copies a unique device identification into **id** based on the information in the VPD page 0x83 of the device. The string will be formatted as a SCSI name string.

Returns the length of the identification or error on failure. If the identifier is longer than the supplied buffer the actual identifier length is returned and the buffer is not zero-padded.

drivers/scsi/scsi_lib_dma.c

SCSI library functions depending on DMA (map and unmap scatter-gather lists).

int **scsi_dma_map**(struct scsi_cmnd * cmd)
perform DMA mapping against command's sg lists

Parameters

struct scsi_cmnd * cmd scsi command

Description

Returns the number of sg lists actually used, zero if the sg lists is NULL, or -ENOMEM if the mapping failed.

void **scsi_dma_unmap**(struct scsi_cmnd * cmd)
unmap command' s sg lists mapped by scsi_dma_map

Parameters

struct scsi_cmnd * cmd scsi command

drivers/scsi/scsi_proc.c

The functions in this file provide an interface between the PROC file system and the SCSI device drivers It is mainly used for debugging, statistics and to pass information directly to the lowlevel driver. I.E. plumbing to manage /proc/scsi/*

void **scsi_proc_hostdir_add**(struct scsi_host_template * sht)
Create directory in /proc for a scsi host

Parameters

struct scsi_host_template * sht owner of this directory

Description

Sets sht->proc_dir to the new directory.

void **scsi_proc_hostdir_rm**(struct scsi_host_template * sht)
remove directory in /proc for a scsi host

Parameters

struct scsi_host_template * sht owner of directory

void **scsi_proc_host_add**(struct Scsi_Host * shost)
Add entry for this host to appropriate /proc dir

Parameters

struct Scsi_Host * shost host to add

void **scsi_proc_host_rm**(struct Scsi_Host * shost)
remove this host' s entry from /proc

Parameters

struct Scsi_Host * shost which host

int **proc_print_scsidevice**(struct device * dev, void * data)
return data about this host

Parameters

struct device * dev A scsi device

void * data `struct seq_file` to output to.

Description

prints Host, Channel, Id, Lun, Vendor, Model, Rev, Type, and revision.

int **scsi_add_single_device**(uint host, uint channel, uint id, uint lun)
Respond to user request to probe for/add device

Parameters

uint host user-supplied decimal integer

uint channel user-supplied decimal integer

uint id user-supplied decimal integer

uint lun user-supplied decimal integer

Description

called by writing "scsi add-single-device" to /proc/scsi/scsi.

does `scsi_host_lookup()` and either user_scan() if that transport type supports it, or else scsi_scan_host_selected()

Note

this seems to be aimed exclusively at SCSI parallel busses.

int **scsi_remove_single_device**(uint host, uint channel, uint id, uint lun)
Respond to user request to remove a device

Parameters

uint host user-supplied decimal integer

uint channel user-supplied decimal integer

uint id user-supplied decimal integer

uint lun user-supplied decimal integer

Description

called by writing "scsi remove-single-device" to /proc/scsi/scsi. Does a `scsi_device_lookup()` and `scsi_remove_device()`

ssize_t **proc_scsi_write**(struct file * file, const char __user * buf,
size_t length, loff_t * ppos)
handle writes to /proc/scsi/scsi

Parameters

struct file * file not used

const char __user * buf buffer to write

size_t length length of buf, at most PAGE_SIZE

loff_t * ppos not used

Description

this provides a legacy mechanism to add or remove devices by Host, Channel, ID, and Lun. To use, "echo 'scsi add-single-device 0 1 2 3' > /proc/scsi/scsi" or "echo 'scsi remove-single-device 0 1 2 3' > /proc/scsi/scsi" with "0 1 2 3" replaced by the Host, Channel, Id, and Lun.

Note

this seems to be aimed at parallel SCSI. Most modern busses (USB, SATA, Firewire, Fibre Channel, etc) dynamically assign these values to provide a unique identifier and nothing more.

int **proc_scsi_open**(struct inode * inode, struct file * file)
glue function

Parameters

struct inode * inode not used

struct file * file passed to single_open()

Description

Associates proc_scsi_show with this file

int **scsi_init_procfs**(void)
create scsi and scsi/scsi in procfs

Parameters

void no arguments

void **scsi_exit_procfs**(void)
Remove scsi/scsi and scsi from procfs

Parameters

void no arguments

drivers/scsi/scsi_netlink.c

Infrastructure to provide async events from transports to userspace via netlink, using a single NETLINK_SCSITRANSPORT protocol for all transports. See the original patch submission for more details.

void **scsi_nl_rcv_msg**(struct sk_buff * skb)
Receive message handler.

Parameters

struct sk_buff * skb socket receive buffer

Description

Extracts message from a receive buffer. Validates message header and calls appropriate transport message handler

void **scsi_netlink_init**(void)
Called by SCSI subsystem to initialize the SCSI transport netlink interface

Parameters

void no arguments

void **scsi_netlink_exit**(void)
Called by SCSI subsystem to disable the SCSI transport netlink interface

Parameters

void no arguments

drivers/scsi/scsi_scan.c

Scan a host to determine which (if any) devices are attached. The general scanning/probing algorithm is as follows, exceptions are made to it depending on device specific flags, compilation options, and global variable (boot or module load time) settings. A specific LUN is scanned via an INQUIRY command; if the LUN has a device attached, a scsi_device is allocated and setup for it. For every id of every channel on the given host, start by scanning LUN 0. Skip hosts that don't respond at all to a scan of LUN 0. Otherwise, if LUN 0 has a device attached, allocate and setup a scsi_device for it. If target is SCSI-3 or up, issue a REPORT LUN, and scan all of the LUNs returned by the REPORT LUN; else, sequentially scan LUNs up until some maximum is reached, or a LUN is seen that cannot have a device attached to it.

int **scsi_complete_async_scans**(void)
Wait for asynchronous scans to complete

Parameters

void no arguments

Description

When this function returns, any host which started scanning before this function was called will have finished its scan. Hosts which started scanning after this function was called may or may not have finished.

void **scsi_unlock_floptical**(struct scsi_device * sdev, unsigned char * result)
unlock device via a special MODE SENSE command

Parameters

struct scsi_device * sdev scsi device to send command to

unsigned char * result area to store the result of the MODE SENSE

Description

Send a vendor specific MODE SENSE (not a MODE SELECT) command. Called for BLIST_KEY devices.

struct scsi_device * **scsi_alloc_sdev**(struct scsi_target * starget, u64 lun, void * hostdata)
allocate and setup a scsi_Device

Parameters

struct scsi_target * starget which target to allocate a `scsi_device` for

u64 lun which lun

void * hostdata usually NULL and set by ->slave_alloc instead

Description

Allocate, initialize for io, and return a pointer to a scsi_Device. Stores the **shost**, **channel**, **id**, and **lun** in the scsi_Device, and adds scsi_Device to the appropriate list.

Return value: scsi_Device pointer, or NULL on failure.

void **scsi_target_reap_ref_release**(struct kref * kref)
remove target from visibility

Parameters

struct kref * kref the reap_ref in the target being released

Description

Called on last put of reap_ref, which is the indication that no device under this target is visible anymore, so render the target invisible in sysfs. Note: we have to be in user context here because the target reaps should be done in places where the scsi device visibility is being removed.

struct scsi_target * **scsi_alloc_target**(struct device * parent, int channel, uint id)
allocate a new or find an existing target

Parameters

struct device * parent parent of the target (need not be a scsi host)

int channel target channel number (zero if no channels)

uint id target id number

Description

Return an existing target if one exists, provided it hasn't already gone into STARGET_DEL state, otherwise allocate a new target.

The target is returned with an incremented reference, so the caller is responsible for both reaping and doing a last put

void **scsi_target_reap**(struct scsi_target * starget)
check to see if target is in use and destroy if not

Parameters

struct scsi_target * starget target to be checked

Description

This is used after removing a LUN or doing a last put of the target it checks atomically that nothing is using the target and removes it if so.

int **scsi_probe_lun**(struct scsi_device * sdev, unsigned char * inq_result, int result_len, blist_flags_t * bflags)
probe a single LUN using a SCSI INQUIRY

Parameters

struct scsi_device * sdev scsi_device to probe

unsigned char * inq_result area to store the INQUIRY result

int result_len len of inq_result

blist_flags_t * bflags store any bflags found here

Description

Probe the lun associated with **req** using a standard SCSI INQUIRY;

If the INQUIRY is successful, zero is returned and the INQUIRY data is in **inq_result**; the scsi_level and INQUIRY length are copied to the scsi_device any flags value is stored in ***bflags**.

int **scsi_add_lun**(struct scsi_device * sdev, unsigned char * inq_result, blist_flags_t * bflags, int async)
allocate and fully initialze a scsi_device

Parameters

struct scsi_device * sdev holds information to be stored in the new scsi_device

unsigned char * inq_result holds the result of a previous INQUIRY to the LUN

blist_flags_t * bflags black/white list flag

int async 1 if this device is being scanned asynchronously

Description

Initialize the scsi_device **sdev**. Optionally set fields based on values in ***bflags**.

Return

SCSI_SCAN_NO_RESPONSE: could not allocate or setup a scsi_device SCSI_SCAN_LUN_PRESENT: a new scsi_device was allocated and initialized

unsigned char * **scsi_inq_str**(unsigned char * buf, unsigned char * inq, unsigned first, unsigned end)
print INQUIRY data from min to max index, strip trailing whitespace

Parameters

unsigned char * buf Output buffer with at least end-first+1 bytes of space

unsigned char * inq Inquiry buffer (input)

unsigned first Offset of string into inq

unsigned end Index after last character in inq

int **scsi_probe_and_add_lun**(struct scsi_target * starget, u64 lun, blist_flags_t * bflagsp, struct scsi_device ** sdevp, enum scsi_scan_mode rescan, void * hostdata)
probe a LUN, if a LUN is found add it

Parameters

struct scsi_target * starget pointer to target device structure

u64 lun LUN of target device

blist_flags_t * bflagsp store bflags here if not NULL

struct scsi_device ** sdevp probe the LUN corresponding to this scsi_device

enum scsi_scan_mode rescan if not equal to SCSI_SCAN_INITIAL skip some code only needed on first scan

void * hostdata passed to `scsi_alloc_sdev()`

Description

Call scsi_probe_lun, if a LUN with an attached device is found, allocate and set it up by calling scsi_add_lun.

Return

- SCSI_SCAN_NO_RESPONSE: could not allocate or setup a scsi_device
- **SCSI_SCAN_TARGET_PRESENT: target responded, but no device is** attached at the LUN
- SCSI_SCAN_LUN_PRESENT: a new scsi_device was allocated and initialized

void **scsi_sequential_lun_scan**(struct scsi_target * starget, blist_flags_t bflags, int scsi_level, enum scsi_scan_mode rescan)
sequentially scan a SCSI target

Parameters

struct scsi_target * starget pointer to target structure to scan

blist_flags_t bflags black/white list flag for LUN 0

int scsi_level Which version of the standard does this device adhere to

enum scsi_scan_mode rescan passed to scsi_probe_add_lun()

Description

Generally, scan from LUN 1 (LUN 0 is assumed to already have been scanned) to some maximum lun until a LUN is found with no device attached. Use the bflags to figure out any oddities.

Modifies sdevscan->lun.

int **scsi_report_lun_scan**(struct scsi_target * starget, blist_flags_t bflags, enum scsi_scan_mode rescan)
Scan using SCSI REPORT LUN results

Parameters

struct scsi_target * starget which target

blist_flags_t bflags Zero or a mix of BLIST_NOLUN, BLIST_REPORTLUN2, or BLIST_NOREPORTLUN

enum scsi_scan_mode rescan nonzero if we can skip code only needed on first scan

Description

Fast scanning for modern (SCSI-3) devices by sending a REPORT LUN command. Scan the resulting list of LUNs by calling scsi_probe_and_add_lun.

If BLINK_REPORTLUN2 is set, scan a target that supports more than 8 LUNs even if it's older than SCSI-3. If BLIST_NOREPORTLUN is set, return 1 always. If BLIST_NOLUN is set, return 0 always. If starget->no_report_luns is set, return 1 always.

Return

0: scan completed (or no memory, so further scanning is futile) 1: could not scan with REPORT LUN

struct async_scan_data * **scsi_prep_async_scan**(struct Scsi_Host * shost)
prepare for an async scan

Parameters

struct Scsi_Host * shost the host which will be scanned

Return

a cookie to be passed to `scsi_finish_async_scan()`

Description

Tells the midlayer this host is going to do an asynchronous scan. It reserves the host's position in the scanning list and ensures that other asynchronous scans started after this one won't affect the ordering of the discovered devices.

void **scsi_finish_async_scan**(struct async_scan_data * data)
asynchronous scan has finished

Parameters

struct async_scan_data * data cookie returned from earlier call to `scsi_prep_async_scan()`

Description

All the devices currently attached to this host have been found. This function announces all the devices it has found to the rest of the system.

drivers/scsi/scsi_sysctl.c

Set up the sysctl entry: "/dev/scsi/logging_level" (DEV_SCSI_LOGGING_LEVEL) which sets/returns scsi_logging_level.

drivers/scsi/scsi_sysfs.c

SCSI sysfs interface routines.

void **scsi_remove_device**(struct scsi_device * sdev)
unregister a device from the scsi bus

Parameters

struct scsi_device * sdev scsi_device to unregister

void **scsi_remove_target**(struct device * dev)
try to remove a target and all its devices

Parameters

struct device * dev generic starget or parent of generic stargets to be removed

Note

This is slightly racy. It is possible that if the user requests the addition of another device then the target won't be removed.

drivers/scsi/hosts.c

mid to lowlevel SCSI driver interface

void **scsi_remove_host**(struct Scsi_Host * shost)
remove a scsi host

Parameters

struct Scsi_Host * shost a pointer to a scsi host to remove

int **scsi_add_host_with_dma**(struct Scsi_Host * shost, struct device * dev, struct device * dma_dev)
add a scsi host with dma device

Parameters

struct Scsi_Host * shost scsi host pointer to add

struct device * dev a struct device of type scsi class

struct device * dma_dev dma device for the host

Note

You rarely need to worry about this unless you' re in a virtualised host environments, so use the simpler scsi_add_host() function instead.

Description

Return value: 0 on success / != 0 for error

struct Scsi_Host * **scsi_host_alloc**(struct scsi_host_template * sht, int privsize)
register a scsi host adapter instance.

Parameters

struct scsi_host_template * sht pointer to scsi host template

int privsize extra bytes to allocate for driver

Note

Allocate a new Scsi_Host and perform basic initialization. The host is not published to the scsi midlayer until scsi_add_host is called.

Description

Return value: Pointer to a new Scsi_Host

struct Scsi_Host * **scsi_host_lookup**(unsigned short hostnum)
get a reference to a Scsi_Host by host no

Parameters

unsigned short hostnum host number to locate

Description

Return value: A pointer to located Scsi_Host or NULL.

The caller must do a `scsi_host_put()` to drop the reference that `scsi_host_get()` took. The `put_device()` below dropped the reference from `class_find_device()`.

struct Scsi_Host * **scsi_host_get**(struct Scsi_Host * shost)
inc a Scsi_Host ref count

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host to inc.

int **scsi_host_busy**(struct Scsi_Host * shost)
Return the host busy counter

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host to inc.

void **scsi_host_put**(struct Scsi_Host * shost)
dec a Scsi_Host ref count

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host to dec.

int **scsi_queue_work**(struct Scsi_Host * shost, struct work_struct * work)
Queue work to the Scsi_Host workqueue.

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host.

struct work_struct * work Work to queue for execution.

Description

Return value: 1 - work queued for execution 0 - work is already queued -EINVAL - work queue doesn't exist

void **scsi_flush_work**(struct Scsi_Host * shost)
Flush a Scsi_Host's workqueue.

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host.

void **scsi_host_complete_all_commands**(struct Scsi_Host * shost, int status)
Terminate all running commands

Parameters

struct Scsi_Host * shost Scsi Host on which commands should be terminated

int status Status to be set for the terminated commands

Description

There is no protection against modification of the number of outstanding commands. It is the responsibility of the caller to ensure that concurrent I/O submission and/or completion is stopped when calling this function.

void **scsi_host_busy_iter**(struct Scsi_Host * shost, bool (*fn)(struct scsi_cmnd *, void *, bool), void * priv)
Iterate over all busy commands

Parameters

struct Scsi_Host * shost Pointer to Scsi_Host.

bool (*)(struct scsi_cmnd *, void *, bool) fn Function to call on each busy command

void * priv Data pointer passed to **fn**

Description

If locking against concurrent command completions is required ithas to be provided by the caller

drivers/scsi/scsi_common.c

general support functions

const char * **scsi_device_type**(unsigned type)
Return 17-char string indicating device type.

Parameters

unsigned type type number to look up

u64 **scsilun_to_int**(struct scsi_lun * scsilun)
convert a scsi_lun to an int

Parameters

struct scsi_lun * scsilun struct scsi_lun to be converted.

Description

Convert **scsilun** from a struct scsi_lun to a four-byte host byte-ordered integer, and return the result. The caller must check for truncation before using this function.

Notes

For a description of the LUN format, post SCSI-3 see the SCSI Architecture Model, for SCSI-3 see the SCSI Controller Commands.

Given a struct scsi_lun of: d2 04 0b 03 00 00 00 00, this function returns the integer: 0x0b03d204

This encoding will return a standard integer LUN for LUNs smaller than 256, which typically use a single level LUN structure with addressing method 0.

void **int_to_scsilun**(u64 lun, struct scsi_lun * scsilun)
reverts an int into a scsi_lun

Parameters

u64 lun integer to be reverted

struct scsi_lun * scsilun struct scsi_lun to be set.

Description

Reverts the functionality of the scsilun_to_int, which packed an 8-byte lun value into an int. This routine unpacks the int back into the lun value.

Notes

Given an integer : 0x0b03d204, this function returns a struct scsi_lun of: d2 04 0b 03 00 00 00 00

bool **scsi_normalize_sense**(const u8 * sense_buffer, int sb_len, struct scsi_sense_hdr * sshdr)
normalize main elements from either fixed or descriptor sense data format into a common format.

Parameters

const u8 * sense_buffer byte array containing sense data returned by device

int sb_len number of valid bytes in sense_buffer

struct scsi_sense_hdr * sshdr pointer to instance of structure that common elements are written to.

Notes

The “main elements” from sense data are: response_code, sense_key, asc, ascq and additional_length (only for descriptor format).

Typically this function can be called after a device has responded to a SCSI command with the CHECK_CONDITION status.

Description

Return value: true if valid sense data information found, else false;

const u8 * **scsi_sense_desc_find**(const u8 * sense_buffer, int sb_len, int desc_type)
search for a given descriptor type in descriptor sense data format.

Parameters

const u8 * sense_buffer byte array of descriptor format sense data

int sb_len number of valid bytes in sense_buffer

int desc_type value of descriptor type to find (e.g. 0 -> information)

Notes

only valid when sense data is in descriptor format

Description

Return value: pointer to start of (first) descriptor if found else NULL

void **scsi_build_sense_buffer**(int desc, u8 * buf, u8 key, u8 asc, u8 ascq)
build sense data in a buffer

Parameters

int desc Sense format (non-zero == descriptor format, 0 == fixed format)

u8 * buf Where to build sense data

u8 key Sense key

u8 asc Additional sense code

u8 ascq Additional sense code qualifier

int **scsi_set_sense_information**(u8 * buf, int buf_len, u64 info)
set the information field in a formatted sense data buffer

Parameters

u8 * buf Where to build sense data

int buf_len buffer length

u64 info 64-bit information value to be set

Description

Return value: 0 on success or -EINVAL for invalid sense buffer length

int **scsi_set_sense_field_pointer**(u8 * buf, int buf_len, u16 fp, u8 bp, bool cd)
set the field pointer sense key specific information in a formatted sense data buffer

Parameters

u8 * buf Where to build sense data

int buf_len buffer length

u16 fp field pointer to be set

u8 bp bit pointer to be set

bool cd command/data bit

Description

Return value: 0 on success or -EINVAL for invalid sense buffer length

32.3.2 Transport classes

Transport classes are service libraries for drivers in the SCSI lower layer, which expose transport attributes in sysfs.

Fibre Channel transport

The file drivers/scsi/scsi_transport_fc.c defines transport attributes for Fibre Channel.

u32 **fc_get_event_number**(void)
Obtain the next sequential FC event number

Parameters

void no arguments

Notes

We could have inlined this, but it would have required fc_event_seq to be exposed. For now, live with the subroutine call. Atomic used to avoid lock/unlock…

void **fc_host_post_fc_event**(struct Scsi_Host * shost, u32 event_number, enum fc_host_event_code event_code, u32 data_len, char * data_buf, u64 vendor_id)

routine to do the work of posting an event on an fc_host.

Parameters

struct Scsi_Host * shost host the event occurred on

u32 event_number fc event number obtained from get_fc_event_number()

enum fc_host_event_code event_code fc_host event being posted

u32 data_len amount, in bytes, of event data

char * data_buf pointer to event data

u64 vendor_id value for Vendor id

Notes

This routine assumes no locks are held on entry.

void **fc_host_post_event**(struct Scsi_Host * shost, u32 event_number, enum fc_host_event_code event_code, u32 event_data)

called to post an even on an fc_host.

Parameters

struct Scsi_Host * shost host the event occurred on

u32 event_number fc event number obtained from get_fc_event_number()

enum fc_host_event_code event_code fc_host event being posted

u32 event_data 32bits of data for the event being posted

Notes

This routine assumes no locks are held on entry.

void **fc_host_post_vendor_event**(struct Scsi_Host * shost, u32 event_number, u32 data_len, char * data_buf, u64 vendor_id)

called to post a vendor unique event on an fc_host

Parameters

struct Scsi_Host * shost host the event occurred on

u32 event_number fc event number obtained from get_fc_event_number()

u32 data_len amount, in bytes, of vendor unique data

char * data_buf pointer to vendor unique data

u64 vendor_id Vendor id

Notes

This routine assumes no locks are held on entry.

void **fc_host_fpin_rcv**(struct Scsi_Host * shost, u32 fpin_len, char * fpin_buf)
routine to process a received FPIN.

Parameters

struct Scsi_Host * shost host the FPIN was received on

u32 fpin_len length of FPIN payload, in bytes

char * fpin_buf pointer to FPIN payload

Notes

This routine assumes no locks are held on entry.

enum blk_eh_timer_return **fc_eh_timed_out**(struct scsi_cmnd * scmd)
FC Transport I/O timeout intercept handler

Parameters

struct scsi_cmnd * scmd The SCSI command which timed out

Description

This routine protects against error handlers getting invoked while a rport is in a blocked state, typically due to a temporarily loss of connectivity. If the error handlers are allowed to proceed, requests to abort i/o, reset the target, etc will likely fail as there is no way to communicate with the device to perform the requested function. These failures may result in the midlayer taking the device offline, requiring manual intervention to restore operation.

This routine, called whenever an i/o times out, validates the state of the underlying rport. If the rport is blocked, it returns EH_RESET_TIMER, which will continue to reschedule the timeout. Eventually, either the device will return, or devloss_tmo will fire, and when the timeout then fires, it will be handled normally. If the rport is not blocked, normal error handling continues.

Notes

This routine assumes no locks are held on entry.

void **fc_remove_host**(struct Scsi_Host * shost)
called to terminate any fc_transport-related elements for a scsi host.

Parameters

struct Scsi_Host * shost Which `Scsi_Host`

Description

This routine is expected to be called immediately preceding the a driver's call to `scsi_remove_host()`.

WARNING: A driver utilizing the fc_transport, which fails to call this routine prior to `scsi_remove_host()`, will leave dangling objects in /sys/class/fc_remote_ports. Access to any of these objects can result in a system crash !!!

Notes

This routine assumes no locks are held on entry.

struct fc_rport * **fc_remote_port_add**(struct Scsi_Host * shost, int channel, struct fc_rport_identifiers * ids)
notify fc transport of the existence of a remote FC port.

Parameters

struct Scsi_Host * shost scsi host the remote port is connected to.

int channel Channel on shost port connected to.

struct fc_rport_identifiers * ids The world wide names, fc address, and FC4 port roles for the remote port.

Description

The LLDD calls this routine to notify the transport of the existence of a remote port. The LLDD provides the unique identifiers (wwpn,wwn) of the port, it's FC address (port_id), and the FC4 roles that are active for the port.

For ports that are FCP targets (aka scsi targets), the FC transport maintains consistent target id bindings on behalf of the LLDD. A consistent target id binding is an assignment of a target id to a remote port identifier, which persists while the scsi host is attached. The remote port can disappear, then later reappear, and it's target id assignment remains the same. This allows for shifts in FC addressing (if binding by wwpn or wwnn) with no apparent changes to the scsi subsystem which is based on scsi host number and target id values. Bindings are only valid during the attachment of the scsi host. If the host detaches, then later re-attaches, target id bindings may change.

This routine is responsible for returning a remote port structure. The routine will search the list of remote ports it maintains internally on behalf of consistent target id mappings. If found, the remote port structure will be reused. Otherwise, a new remote port structure will be allocated.

Whenever a remote port is allocated, a new fc_remote_port class device is created.

Should not be called from interrupt context.

Notes

This routine assumes no locks are held on entry.

void **fc_remote_port_delete**(struct fc_rport * rport)
notifies the fc transport that a remote port is no longer in existence.

Parameters

struct fc_rport * rport The remote port that no longer exists

Description

The LLDD calls this routine to notify the transport that a remote port is no longer part of the topology. Note: Although a port may no longer be part of the topology, it may persist in the remote ports displayed by the fc_host. We do this under 2 conditions:

1) If the port was a scsi target, we delay its deletion by "blocking" it. This allows the port to temporarily disappear, then reappear without disrupting the SCSI device tree attached to it. During the "blocked" period the port will still exist.

2) If the port was a scsi target and disappears for longer than we expect, we'll delete the port and the tear down the SCSI device tree attached to it. However, we want to semi-persist the target id assigned to that port if it eventually does exist. The port structure will remain (although with minimal information) so that the target id bindings also remain.

If the remote port is not an FCP Target, it will be fully torn down and deallocated, including the fc_remote_port class device.

If the remote port is an FCP Target, the port will be placed in a temporary blocked state. From the LLDD's perspective, the rport no longer exists. From the SCSI midlayer's perspective, the SCSI target exists, but all sdevs on it are blocked from further I/O. The following is then expected.

If the remote port does not return (signaled by a LLDD call to `fc_remote_port_add()`) within the dev_loss_tmo timeout, then the scsi target is removed - killing all outstanding i/o and removing the scsi devices attached to it. The port structure will be marked Not Present and be partially cleared, leaving only enough information to recognize the remote port relative to the scsi target id binding if it later appears. The port will remain as long as there is a valid binding (e.g. until the user changes the binding type or unloads the scsi host with the binding).

If the remote port returns within the dev_loss_tmo value (and matches according to the target id binding type), the port structure will be reused. If it is no longer a SCSI target, the target will be torn down. If it continues to be a SCSI target, then the target will be unblocked (allowing i/o to be resumed), and a scan will be activated to ensure that all luns are detected.

Called from normal process context only - cannot be called from interrupt.

Notes

This routine assumes no locks are held on entry.

void **fc_remote_port_rolechg**(struct fc_rport * rport, u32 roles)
notifies the fc transport that the roles on a remote may have changed.

Parameters

struct fc_rport * rport The remote port that changed.

u32 roles New roles for this port.

Description

The LLDD calls this routine to notify the transport that the roles on a remote port may have changed. The largest effect of this is if a port now becomes a FCP Target, it must be allocated a scsi target id. If the port is no longer a FCP target, any scsi target id value assigned to it will persist in case the role changes back to include FCP Target. No changes in the scsi midlayer will be invoked if the role changes (in the expectation that the role will be resumed. If it doesn't normal error processing will take place).

Should not be called from interrupt context.

Notes

This routine assumes no locks are held on entry.

int **fc_block_rport**(struct fc_rport * rport)
Block SCSI eh thread for blocked fc_rport.

Parameters

struct fc_rport * rport Remote port that scsi_eh is trying to recover.

Description

This routine can be called from a FC LLD scsi_eh callback. It blocks the scsi_eh thread until the fc_rport leaves the FC_PORTSTATE_BLOCKED, or the fast_io_fail_tmo fires. This is necessary to avoid the scsi_eh failing recovery actions for blocked rports which would lead to offlined SCSI devices.

Return

0 if the fc_rport left the state FC_PORTSTATE_BLOCKED. FAST_IO_FAIL if the fast_io_fail_tmo fired, this should be passed back to scsi_eh.

int **fc_block_scsi_eh**(struct scsi_cmnd * cmnd)
Block SCSI eh thread for blocked fc_rport

Parameters

struct scsi_cmnd * cmnd SCSI command that scsi_eh is trying to recover

Description

This routine can be called from a FC LLD scsi_eh callback. It blocks the scsi_eh thread until the fc_rport leaves the FC_PORTSTATE_BLOCKED, or the fast_io_fail_tmo fires. This is necessary to avoid the scsi_eh failing recovery actions for blocked rports which would lead to offlined SCSI devices.

Return

0 if the fc_rport left the state FC_PORTSTATE_BLOCKED. FAST_IO_FAIL if the fast_io_fail_tmo fired, this should be passed back to scsi_eh.

struct fc_vport * **fc_vport_create**(struct Scsi_Host * shost, int channel,
struct fc_vport_identifiers * ids)
Admin App or LLDD requests creation of a vport

Parameters

struct Scsi_Host * shost scsi host the virtual port is connected to.

int channel channel on shost port connected to.

struct fc_vport_identifiers * ids The world wide names, FC4 port roles, etc for the virtual port.

Notes

This routine assumes no locks are held on entry.

int **fc_vport_terminate**(struct fc_vport * vport)
Admin App or LLDD requests termination of a vport

Parameters

struct fc_vport * vport fc_vport to be terminated

Description

Calls the LLDD vport_delete() function, then deallocates and removes the vport from the shost and object tree.

Notes

This routine assumes no locks are held on entry.

iSCSI transport class

The file drivers/scsi/scsi_transport_iscsi.c defines transport attributes for the iSCSI class, which sends SCSI packets over TCP/IP connections.

struct iscsi_bus_flash_session * **iscsi_create_flashnode_sess**(struct Scsi_Host * shost, int index, struct iscsi_transport * transport, int dd_size)

Add flashnode session entry in sysfs

Parameters

struct Scsi_Host * shost pointer to host data

int index index of flashnode to add in sysfs

struct iscsi_transport * transport pointer to transport data

int dd_size total size to allocate

Description

Adds a sysfs entry for the flashnode session attributes

Return

pointer to allocated flashnode sess on success NULL on failure

struct iscsi_bus_flash_conn * **iscsi_create_flashnode_conn**(struct Scsi_Host * shost, struct iscsi_bus_flash_session * fnode_sess, struct iscsi_transport * transport, int dd_size)

Add flashnode conn entry in sysfs

Parameters

struct Scsi_Host * shost pointer to host data

struct iscsi_bus_flash_session * fnode_sess pointer to the parent flashnode session entry

struct iscsi_transport * transport pointer to transport data

int dd_size total size to allocate

Description

Adds a sysfs entry for the flashnode connection attributes

Return

pointer to allocated flashnode conn on success NULL on failure

struct device * **iscsi_find_flashnode_sess**(struct Scsi_Host * shost, void * data, int (*fn)(struct device *dev, void *data))

finds flashnode session entry

Parameters

struct Scsi_Host * shost pointer to host data

void * data pointer to data containing value to use for comparison

int (*)(struct device *dev, void *data) fn function pointer that does actual comparison

Description

Finds the flashnode session object comparing the data passed using logic defined in passed function pointer

Return

pointer to found flashnode session device object on success NULL on failure

struct device * **iscsi_find_flashnode_conn**(struct iscsi_bus_flash_session * fnode_sess)

finds flashnode connection entry

Parameters

struct iscsi_bus_flash_session * fnode_sess pointer to parent flashnode session entry

Description

Finds the flashnode connection object comparing the data passed using logic defined in passed function pointer

Return

pointer to found flashnode connection device object on success NULL on failure

void **iscsi_destroy_flashnode_sess**(struct iscsi_bus_flash_session * fnode_sess)

destroy flashnode session entry

Parameters

struct iscsi_bus_flash_session * fnode_sess pointer to flashnode session entry to be destroyed

Description

Deletes the flashnode session entry and all children flashnode connection entries from sysfs

void **iscsi_destroy_all_flashnode**(struct Scsi_Host * shost)
destroy all flashnode session entries

Parameters

struct Scsi_Host * shost pointer to host data

Description

Destroys all the flashnode session entries and all corresponding children flashnode connection entries from sysfs

int **iscsi_scan_finished**(struct Scsi_Host * shost, unsigned long time)
helper to report when running scans are done

Parameters

struct Scsi_Host * shost scsi host

unsigned long time scan run time

Description

This function can be used by drives like qla4xxx to report to the scsi layer when the scans it kicked off at module load time are done.

int **iscsi_block_scsi_eh**(struct scsi_cmnd * cmd)
block scsi eh until session state has transistioned

Parameters

struct scsi_cmnd * cmd scsi cmd passed to scsi eh handler

Description

If the session is down this function will wait for the recovery timer to fire or for the session to be logged back in. If the recovery timer fires then FAST_IO_FAIL is returned. The caller should pass this error value to the scsi eh.

void **iscsi_unblock_session**(struct iscsi_cls_session * session)
set a session as logged in and start IO.

Parameters

struct iscsi_cls_session * session iscsi session

Description

Mark a session as ready to accept IO.

struct iscsi_cls_session * **iscsi_create_session**(struct Scsi_Host * shost, struct iscsi_transport * transport, int dd_size, unsigned int target_id)
create iscsi class session

Parameters

struct Scsi_Host * shost scsi host

struct iscsi_transport * transport iscsi transport

int dd_size private driver data size

unsigned int target_id which target

Description

This can be called from a LLD or iscsi_transport.

struct iscsi_cls_conn * **iscsi_create_conn**(struct iscsi_cls_session * session, int dd_size, uint32_t cid)
create iscsi class connection

Parameters

struct iscsi_cls_session * session iscsi cls session

int dd_size private driver data size

uint32_t cid connection id

Description

This can be called from a LLD or iscsi_transport. The connection is child of the session so cid must be unique for all connections on the session.

Since we do not support MCS, cid will normally be zero. In some cases for software iscsi we could be trying to preallocate a connection struct in which case there could be two connection structs and cid would be non-zero.

int **iscsi_destroy_conn**(struct iscsi_cls_conn * conn)
destroy iscsi class connection

Parameters

struct iscsi_cls_conn * conn iscsi cls session

Description

This can be called from a LLD or iscsi_transport.

int **iscsi_session_event**(struct iscsi_cls_session * session, enum iscsi_uevent_e event)
send session destr. completion event

Parameters

struct iscsi_cls_session * session iscsi class session

enum iscsi_uevent_e event type of event

Serial Attached SCSI (SAS) transport class

The file drivers/scsi/scsi_transport_sas.c defines transport attributes for Serial Attached SCSI, a variant of SATA aimed at large high-end systems.

The SAS transport class contains common code to deal with SAS HBAs, an aproximated representation of SAS topologies in the driver model, and various sysfs attributes to expose these topologies and management interfaces to userspace.

In addition to the basic SCSI core objects this transport class introduces two additional intermediate objects: The SAS PHY as represented by struct sas_phy defines an "outgoing" PHY on a SAS HBA or Expander, and the SAS remote PHY represented by struct sas_rphy defines an "incoming" PHY on a SAS Expander or end device. Note that this is purely a software concept, the underlying hardware for a PHY and a remote PHY is the exactly the same.

There is no concept of a SAS port in this code, users can see what PHYs form a wide port based on the port_identifier attribute, which is the same for all PHYs in a port.

void **sas_remove_children**(struct device * dev)
tear down a devices SAS data structures

Parameters

struct device * dev device belonging to the sas object

Description

Removes all SAS PHYs and remote PHYs for a given object

void **sas_remove_host**(struct Scsi_Host * shost)
tear down a Scsi_Host's SAS data structures

Parameters

struct Scsi_Host * shost Scsi Host that is torn down

Description

Removes all SAS PHYs and remote PHYs for a given Scsi_Host and remove the Scsi_Host as well.

Note

Do not call `scsi_remove_host()` on the Scsi_Host any more, as it is already removed.

u64 **sas_get_address**(struct scsi_device * sdev)
return the SAS address of the device

Parameters

struct scsi_device * sdev scsi device

Description

Returns the SAS address of the scsi device

unsigned int **sas_tlr_supported**(struct scsi_device * sdev)
checking TLR bit in vpd 0x90

Parameters

struct scsi_device * sdev scsi device struct

Description

Check Transport Layer Retries are supported or not. If vpd page 0x90 is present, TRL is supported.

void **sas_disable_tlr**(struct scsi_device * sdev)
setting TLR flags

Parameters

struct scsi_device * sdev scsi device struct

Description

Seting tlr_enabled flag to 0.

void **sas_enable_tlr**(struct scsi_device * sdev)
setting TLR flags

Parameters

struct scsi_device * sdev scsi device struct

Description

Seting tlr_enabled flag 1.

struct sas_phy * **sas_phy_alloc**(struct device * parent, int number)
allocates and initialize a SAS PHY structure

Parameters

struct device * parent Parent device

int number Phy index

Description

Allocates an SAS PHY structure. It will be added in the device tree below the device specified by **parent**, which has to be either a Scsi_Host or sas_rphy.

Return

SAS PHY allocated or `NULL` if the allocation failed.

int **sas_phy_add**(struct sas_phy * phy)
add a SAS PHY to the device hierarchy

Parameters

struct sas_phy * phy The PHY to be added

Description

Publishes a SAS PHY to the rest of the system.

void **sas_phy_free**(struct sas_phy * phy)
free a SAS PHY

Parameters

struct sas_phy * phy SAS PHY to free

Description

Frees the specified SAS PHY.

Note

This function must only be called on a PHY that has not successfully been added using `sas_phy_add()`.

void **sas_phy_delete**(struct sas_phy * phy)
remove SAS PHY

Parameters

struct sas_phy * phy SAS PHY to remove

Description

Removes the specified SAS PHY. If the SAS PHY has an associated remote PHY it is removed before.

int **scsi_is_sas_phy**(const struct device * dev)
check if a struct device represents a SAS PHY

Parameters

const struct device * dev device to check

Return

`1` if the device represents a SAS PHY, `0` else

int **sas_port_add**(struct sas_port * port)
add a SAS port to the device hierarchy

Parameters

struct sas_port * port port to be added

Description

publishes a port to the rest of the system

void **sas_port_free**(struct sas_port * port)
free a SAS PORT

Parameters

struct sas_port * port SAS PORT to free

Description

Frees the specified SAS PORT.

Note

This function must only be called on a PORT that has not successfully been added using `sas_port_add()`.

void **sas_port_delete**(struct sas_port * port)
remove SAS PORT

Parameters

struct sas_port * port SAS PORT to remove

Description

Removes the specified SAS PORT. If the SAS PORT has an associated phys, unlink them from the port as well.

int **scsi_is_sas_port**(const struct device * dev)
check if a struct device represents a SAS port

Parameters

const struct device * dev device to check

Return

1 if the device represents a SAS Port, 0 else

struct sas_phy * **sas_port_get_phy**(struct sas_port * port)
try to take a reference on a port member

Parameters

struct sas_port * port port to check

void **sas_port_add_phy**(struct sas_port * port, struct sas_phy * phy)
add another phy to a port to form a wide port

Parameters

struct sas_port * port port to add the phy to

struct sas_phy * phy phy to add

Description

When a port is initially created, it is empty (has no phys). All ports must have at least one phy to operated, and all wide ports must have at least two. The current code makes no difference between ports and wide ports, but the only object that can be connected to a remote device is a port, so ports must be formed on all devices with phys if they' re connected to anything.

void **sas_port_delete_phy**(struct sas_port * port, struct sas_phy * phy)
remove a phy from a port or wide port

Parameters

struct sas_port * port port to remove the phy from

struct sas_phy * phy phy to remove

Description

This operation is used for tearing down ports again. It must be done to every port or wide port before calling sas_port_delete.

struct sas_rphy * **sas_end_device_alloc**(struct sas_port * parent)
allocate an rphy for an end device

Parameters

struct sas_port * parent which port

Description

Allocates an SAS remote PHY structure, connected to **parent**.

Return

SAS PHY allocated or `NULL` if the allocation failed.

struct sas_rphy * **sas_expander_alloc**(struct sas_port * parent, enum sas_device_type type)
allocate an rphy for an end device

Parameters

struct sas_port * parent which port

enum sas_device_type type SAS_EDGE_EXPANDER_DEVICE or SAS_FANOUT_EXPANDER_DEVICE

Description

Allocates an SAS remote PHY structure, connected to **parent**.

Return

SAS PHY allocated or `NULL` if the allocation failed.

int **sas_rphy_add**(struct sas_rphy * rphy)
add a SAS remote PHY to the device hierarchy

Parameters

struct sas_rphy * rphy The remote PHY to be added

Description

Publishes a SAS remote PHY to the rest of the system.

void **sas_rphy_free**(struct sas_rphy * rphy)
free a SAS remote PHY

Parameters

struct sas_rphy * rphy SAS remote PHY to free

Description

Frees the specified SAS remote PHY.

Note

This function must only be called on a remote PHY that has not successfully been added using `sas_rphy_add()` (or has been `sas_rphy_remove()`' d)

void **sas_rphy_delete**(struct sas_rphy * rphy)
remove and free SAS remote PHY

Parameters

struct sas_rphy * rphy SAS remote PHY to remove and free

Description

Removes the specified SAS remote PHY and frees it.

void **sas_rphy_unlink**(struct sas_rphy * rphy)
unlink SAS remote PHY

Parameters

struct sas_rphy * rphy SAS remote phy to unlink from its parent port

Description

Removes port reference to an rphy

void **sas_rphy_remove**(struct sas_rphy * rphy)
remove SAS remote PHY

Parameters

struct sas_rphy * rphy SAS remote phy to remove

Description

Removes the specified SAS remote PHY.

int **scsi_is_sas_rphy**(const struct device * dev)
check if a struct device represents a SAS remote PHY

Parameters

const struct device * dev device to check

Return

1 if the device represents a SAS remote PHY, 0 else

struct scsi_transport_template * **sas_attach_transport**(struct sas_function_template * ft)
instantiate SAS transport template

Parameters

struct sas_function_template * ft SAS transport class function template

void **sas_release_transport**(struct scsi_transport_template * t)
release SAS transport template instance

Parameters

struct scsi_transport_template * t transport template instance

SATA transport class

The SATA transport is handled by libata, which has its own book of documentation in this directory.

Parallel SCSI (SPI) transport class

The file drivers/scsi/scsi_transport_spi.c defines transport attributes for traditional (fast/wide/ultra) SCSI busses.

void **spi_schedule_dv_device**(struct scsi_device * sdev)
schedule domain validation to occur on the device

Parameters

struct scsi_device * sdev The device to validate

Identical to spi_dv_device() above, except that the DV will be scheduled to occur in a workqueue later. All memory allocations are atomic, so may be called from any context including those holding SCSI locks.

void **spi_display_xfer_agreement**(struct scsi_target * starget)
Print the current target transfer agreement

Parameters

struct scsi_target * starget The target for which to display the agreement

Description

Each SPI port is required to maintain a transfer agreement for each other port on the bus. This function prints a one-line summary of the current agreement; more detailed information is available in sysfs.

int **spi_populate_tag_msg**(unsigned char * msg, struct scsi_cmnd * cmd)
place a tag message in a buffer

Parameters

unsigned char * msg pointer to the area to place the tag

struct scsi_cmnd * cmd pointer to the scsi command for the tag

Notes

designed to create the correct type of tag message for the particular request. Returns the size of the tag message. May return 0 if TCQ is disabled for this device.

SCSI RDMA (SRP) transport class

The file drivers/scsi/scsi_transport_srp.c defines transport attributes for SCSI over Remote Direct Memory Access.

int **srp_tmo_valid**(int reconnect_delay, int fast_io_fail_tmo,
long dev_loss_tmo)
check timeout combination validity

Parameters

int reconnect_delay Reconnect delay in seconds.

int fast_io_fail_tmo Fast I/O fail timeout in seconds.

long dev_loss_tmo Device loss timeout in seconds.

Description

The combination of the timeout parameters must be such that SCSI commands are finished in a reasonable time. Hence do not allow the fast I/O fail timeout to exceed SCSI_DEVICE_BLOCK_MAX_TIMEOUT nor allow dev_loss_tmo to exceed that limit if failing I/O fast has been disabled. Furthermore, these parameters must be such that multipath can detect failed paths timely. Hence do not allow all three parameters to be disabled simultaneously.

void **srp_start_tl_fail_timers**(struct srp_rport * rport)
start the transport layer failure timers

Parameters

struct srp_rport * rport SRP target port.

Description

Start the transport layer fast I/O failure and device loss timers. Do not modify a timer that was already started.

int **srp_reconnect_rport**(struct srp_rport * rport)
reconnect to an SRP target port

Parameters

struct srp_rport * rport SRP target port.

Description

Blocks SCSI command queueing before invoking reconnect() such that queuecommand() won't be invoked concurrently with reconnect() from outside the SCSI EH. This is important since a reconnect() implementation may reallocate resources needed by queuecommand().

Notes

- This function neither waits until outstanding requests have finished nor tries to abort these. It is the responsibility of the reconnect() function to finish outstanding commands before reconnecting to the target port.
- It is the responsibility of the caller to ensure that the resources reallocated by the reconnect() function won't be used while this function is in progress. One possible strategy is to invoke this function from the context of the SCSI EH thread only. Another possible strategy is to lock the rport mutex inside each SCSI LLD callback that can be invoked by the SCSI EH (the scsi_host_template.eh_*() functions and also the scsi_host_template.queuecommand() function).

enum blk_eh_timer_return **srp_timed_out**(struct scsi_cmnd * scmd)
SRP transport intercept of the SCSI timeout EH

Parameters

struct scsi_cmnd * scmd SCSI command.

Description

If a timeout occurs while an rport is in the blocked state, ask the SCSI EH to continue waiting (BLK_EH_RESET_TIMER). Otherwise let the SCSI core handle the timeout (BLK_EH_DONE).

Note

This function is called from soft-IRQ context and with the request queue lock held.

void **srp_rport_get**(struct srp_rport * rport)
increment rport reference count

Parameters

struct srp_rport * rport SRP target port.

void **srp_rport_put**(struct srp_rport * rport)
decrement rport reference count

Parameters

struct srp_rport * rport SRP target port.

struct srp_rport * **srp_rport_add**(struct Scsi_Host * shost, struct srp_rport_identifiers * ids)
add a SRP remote port to the device hierarchy

Parameters

struct Scsi_Host * shost scsi host the remote port is connected to.

struct srp_rport_identifiers * ids The port id for the remote port.

Description

Publishes a port to the rest of the system.

void **srp_rport_del**(struct srp_rport * rport)
remove a SRP remote port

Parameters

struct srp_rport * rport SRP remote port to remove

Description

Removes the specified SRP remote port.

void **srp_remove_host**(struct Scsi_Host * shost)
tear down a Scsi_Host's SRP data structures

Parameters

struct Scsi_Host * shost Scsi Host that is torn down

Description

Removes all SRP remote ports for a given Scsi_Host. Must be called just before scsi_remove_host for SRP HBAs.

void **srp_stop_rport_timers**(struct srp_rport * rport)
stop the transport layer recovery timers

Parameters

struct srp_rport * rport SRP remote port for which to stop the timers.

Description

Must be called after `srp_remove_host()` and `scsi_remove_host()`. The caller must hold a reference on the rport (rport->dev) and on the SCSI host (rport->dev.parent).

struct scsi_transport_template * **srp_attach_transport**(struct srp_function_template * ft)
instantiate SRP transport template

Parameters

struct srp_function_template * ft SRP transport class function template

void **srp_release_transport**(struct scsi_transport_template * t)
release SRP transport template instance

Parameters

struct scsi_transport_template * t transport template instance

32.4 SCSI lower layer

32.4.1 Host Bus Adapter transport types

Many modern device controllers use the SCSI command set as a protocol to communicate with their devices through many different types of physical connections.

In SCSI language a bus capable of carrying SCSI commands is called a “transport”, and a controller connecting to such a bus is called a “host bus adapter” (HBA).

Debug transport

The file drivers/scsi/scsi_debug.c simulates a host adapter with a variable number of disks (or disk like devices) attached, sharing a common amount of RAM. Does a lot of checking to make sure that we are not getting blocks mixed up, and panics the kernel if anything out of the ordinary is seen.

To be more realistic, the simulated devices have the transport attributes of SAS disks.

For documentation see http://sg.danny.cz/sg/sdebug26.html

todo

Parallel (fast/wide/ultra) SCSI, USB, SATA, SAS, Fibre Channel, FireWire, ATAPI devices, Infiniband, I2O, Parallel ports, netlink…

CHAPTER

THIRTYTHREE

LIBATA DEVELOPER'S GUIDE

Author Jeff Garzik

33.1 Introduction

libATA is a library used inside the Linux kernel to support ATA host controllers and devices. libATA provides an ATA driver API, class transports for ATA and ATAPI devices, and SCSI<->ATA translation for ATA devices according to the T10 SAT specification.

This Guide documents the libATA driver API, library functions, library internals, and a couple sample ATA low-level drivers.

33.2 libata Driver API

`struct ata_port_operations` is defined for every low-level libata hardware driver, and it controls how the low-level driver interfaces with the ATA and SCSI layers.

FIS-based drivers will hook into the system with `->qc_prep()` and `->qc_issue()` high-level hooks. Hardware which behaves in a manner similar to PCI IDE hardware may utilize several generic helpers, defining at a bare minimum the bus I/O addresses of the ATA shadow register blocks.

33.2.1 `struct ata_port_operations`

Disable ATA port

```
void (*port_disable) (struct ata_port *);
```

Called from `ata_bus_probe()` error path, as well as when unregistering from the SCSI module (rmmod, hot unplug). This function should do whatever needs to be done to take the port out of use. In most cases, `ata_port_disable()` can be used as this hook.

Called from `ata_bus_probe()` on a failed probe. Called from `ata_scsi_release()`.

Post-IDENTIFY device configuration

```
void (*dev_config) (struct ata_port *, struct ata_device *);
```

Called after IDENTIFY [PACKET] DEVICE is issued to each device found. Typically used to apply device-specific fixups prior to issue of SET FEATURES - XFER MODE, and prior to operation.

This entry may be specified as NULL in ata_port_operations.

Set PIO/DMA mode

```
void (*set_piomode) (struct ata_port *, struct ata_device *);
void (*set_dmamode) (struct ata_port *, struct ata_device *);
void (*post_set_mode) (struct ata_port *);
unsigned int (*mode_filter) (struct ata_port *, struct ata_device *,␣
↪unsigned int);
```

Hooks called prior to the issue of SET FEATURES - XFER MODE command. The optional `->mode_filter()` hook is called when libata has built a mask of the possible modes. This is passed to the `->mode_filter()` function which should return a mask of valid modes after filtering those unsuitable due to hardware limits. It is not valid to use this interface to add modes.

`dev->pio_mode` and `dev->dma_mode` are guaranteed to be valid when `->set_piomode()` and when `->set_dmamode()` is called. The timings for any other drive sharing the cable will also be valid at this point. That is the library records the decisions for the modes of each drive on a channel before it attempts to set any of them.

`->post_set_mode()` is called unconditionally, after the SET FEATURES - XFER MODE command completes successfully.

`->set_piomode()` is always called (if present), but `->set_dma_mode()` is only called if DMA is possible.

Taskfile read/write

```
void (*sff_tf_load) (struct ata_port *ap, struct ata_taskfile *tf);
void (*sff_tf_read) (struct ata_port *ap, struct ata_taskfile *tf);
```

`->tf_load()` is called to load the given taskfile into hardware registers / DMA buffers. `->tf_read()` is called to read the hardware registers / DMA buffers, to obtain the current set of taskfile register values. Most drivers for taskfile-based hardware (PIO or MMIO) use `ata_sff_tf_load()` and `ata_sff_tf_read()` for these hooks.

PIO data read/write

```
void (*sff_data_xfer) (struct ata_device *, unsigned char *, unsigned int,␣
↪int);
```

All bmdma-style drivers must implement this hook. This is the low-level operation that actually copies the data bytes during a PIO data transfer. Typically the driver will choose one of `ata_sff_data_xfer()`, or `ata_sff_data_xfer32()`.

ATA command execute

```
void (*sff_exec_command)(struct ata_port *ap, struct ata_taskfile *tf);
```

causes an ATA command, previously loaded with `->tf_load()`, to be initiated in hardware. Most drivers for taskfile-based hardware use `ata_sff_exec_command()` for this hook.

Per-cmd ATAPI DMA capabilities filter

```
int (*check_atapi_dma) (struct ata_queued_cmd *qc);
```

Allow low-level driver to filter ATA PACKET commands, returning a status indicating whether or not it is OK to use DMA for the supplied PACKET command.

This hook may be specified as NULL, in which case libata will assume that atapi dma can be supported.

Read specific ATA shadow registers

```
u8   (*sff_check_status)(struct ata_port *ap);
u8   (*sff_check_altstatus)(struct ata_port *ap);
```

Reads the Status/AltStatus ATA shadow register from hardware. On some hardware, reading the Status register has the side effect of clearing the interrupt condition. Most drivers for taskfile-based hardware use `ata_sff_check_status()` for this hook.

Write specific ATA shadow register

```
void (*sff_set_devctl)(struct ata_port *ap, u8 ctl);
```

Write the device control ATA shadow register to the hardware. Most drivers don't need to define this.

Select ATA device on bus

```
void (*sff_dev_select)(struct ata_port *ap, unsigned int device);
```

Issues the low-level hardware command(s) that causes one of N hardware devices to be considered 'selected' (active and available for use) on the ATA bus. This generally has no meaning on FIS-based devices.

Most drivers for taskfile-based hardware use `ata_sff_dev_select()` for this hook.

Private tuning method

```
void (*set_mode) (struct ata_port *ap);
```

By default libata performs drive and controller tuning in accordance with the ATA timing rules and also applies blacklists and cable limits. Some controllers need special handling and have custom tuning rules, typically raid controllers that use ATA commands but do not actually do drive timing.

> **Warning**
>
> This hook should not be used to replace the standard controller tuning logic when a controller has quirks. Replacing the default tuning logic in that case would bypass handling for drive and bridge quirks that may be important to data reliability. If a controller needs to filter the mode selection it should use the mode_filter hook instead.

Control PCI IDE BMDMA engine

```
void (*bmdma_setup) (struct ata_queued_cmd *qc);
void (*bmdma_start) (struct ata_queued_cmd *qc);
void (*bmdma_stop) (struct ata_port *ap);
u8   (*bmdma_status) (struct ata_port *ap);
```

When setting up an IDE BMDMA transaction, these hooks arm (`->bmdma_setup`), fire (`->bmdma_start`), and halt (`->bmdma_stop`) the hardware's DMA engine. `->bmdma_status` is used to read the standard PCI IDE DMA Status register.

These hooks are typically either no-ops, or simply not implemented, in FIS-based drivers.

Most legacy IDE drivers use `ata_bmdma_setup()` for the `bmdma_setup()` hook. `ata_bmdma_setup()` will write the pointer to the PRD table to the IDE PRD Table Address register, enable DMA in the DMA Command register, and call `exec_command()` to begin the transfer.

Most legacy IDE drivers use `ata_bmdma_start()` for the `bmdma_start()` hook. `ata_bmdma_start()` will write the ATA_DMA_START flag to the DMA Command register.

Many legacy IDE drivers use `ata_bmdma_stop()` for the `bmdma_stop()` hook. `ata_bmdma_stop()` clears the ATA_DMA_START flag in the DMA command register.

Many legacy IDE drivers use ata_bmdma_status() as the bmdma_status() hook.

High-level taskfile hooks

```
enum ata_completion_errors (*qc_prep) (struct ata_queued_cmd *qc);
int (*qc_issue) (struct ata_queued_cmd *qc);
```

Higher-level hooks, these two hooks can potentially supersede several of the above taskfile/DMA engine hooks. ->qc_prep is called after the buffers have been DMA-mapped, and is typically used to populate the hardware's DMA scatter-gather table. Some drivers use the standard ata_bmdma_qc_prep() and ata_bmdma_dumb_qc_prep() helper functions, but more advanced drivers roll their own.

->qc_issue is used to make a command active, once the hardware and S/G tables have been prepared. IDE BMDMA drivers use the helper function ata_sff_qc_issue() for taskfile protocol-based dispatch. More advanced drivers implement their own ->qc_issue.

ata_sff_qc_issue() calls ->sff_tf_load(), ->bmdma_setup(), and ->bmdma_start() as necessary to initiate a transfer.

Exception and probe handling (EH)

```
void (*eng_timeout) (struct ata_port *ap);
void (*phy_reset) (struct ata_port *ap);
```

Deprecated. Use ->error_handler() instead.

```
void (*freeze) (struct ata_port *ap);
void (*thaw) (struct ata_port *ap);
```

ata_port_freeze() is called when HSM violations or some other condition disrupts normal operation of the port. A frozen port is not allowed to perform any operation until the port is thawed, which usually follows a successful reset.

The optional ->freeze() callback can be used for freezing the port hardware-wise (e.g. mask interrupt and stop DMA engine). If a port cannot be frozen hardware-wise, the interrupt handler must ack and clear interrupts unconditionally while the port is frozen.

The optional ->thaw() callback is called to perform the opposite of ->freeze(): prepare the port for normal operation once again. Unmask interrupts, start DMA engine, etc.

```
void (*error_handler) (struct ata_port *ap);
```

->error_handler() is a driver's hook into probe, hotplug, and recovery and other exceptional conditions. The primary responsibility of an implementation is to call ata_do_eh() or ata_bmdma_drive_eh() with a set of EH hooks as arguments:

'prereset' hook (may be NULL) is called during an EH reset, before any other actions are taken.

'postreset' hook (may be NULL) is called after the EH reset is performed. Based on existing conditions, severity of the problem, and hardware capabilities,

Either 'softreset' (may be NULL) or 'hardreset' (may be NULL) will be called to perform the low-level EH reset.

```
void (*post_internal_cmd) (struct ata_queued_cmd *qc);
```

Perform any hardware-specific actions necessary to finish processing after executing a probe-time or EH-time command via `ata_exec_internal()`.

Hardware interrupt handling

```
irqreturn_t (*irq_handler)(int, void *, struct pt_regs *);
void (*irq_clear) (struct ata_port *);
```

`->irq_handler` is the interrupt handling routine registered with the system, by libata. `->irq_clear` is called during probe just before the interrupt handler is registered, to be sure hardware is quiet.

The second argument, dev_instance, should be cast to a pointer to `struct ata_host_set`.

Most legacy IDE drivers use `ata_sff_interrupt()` for the irq_handler hook, which scans all ports in the host_set, determines which queued command was active (if any), and calls ata_sff_host_intr(ap,qc).

Most legacy IDE drivers use `ata_sff_irq_clear()` for the `irq_clear()` hook, which simply clears the interrupt and error flags in the DMA status register.

SATA phy read/write

```
int (*scr_read) (struct ata_port *ap, unsigned int sc_reg,
         u32 *val);
int (*scr_write) (struct ata_port *ap, unsigned int sc_reg,
                  u32 val);
```

Read and write standard SATA phy registers. Currently only used if `->phy_reset` hook called the `sata_phy_reset()` helper function. sc_reg is one of SCR_STATUS, SCR_CONTROL, SCR_ERROR, or SCR_ACTIVE.

Init and shutdown

```
int (*port_start) (struct ata_port *ap);
void (*port_stop) (struct ata_port *ap);
void (*host_stop) (struct ata_host_set *host_set);
```

`->port_start()` is called just after the data structures for each port are initialized. Typically this is used to alloc per-port DMA buffers / tables / rings, enable DMA engines, and similar tasks. Some drivers also use this entry point as a chance to allocate driver-private memory for `ap->private_data`.

Many drivers use `ata_port_start()` as this hook or call it from their own `port_start()` hooks. `ata_port_start()` allocates space for a legacy IDE PRD table and returns.

`->port_stop()` is called after `->host_stop()`. Its sole function is to release DMA/memory resources, now that they are no longer actively being used. Many drivers also free driver-private data from port at this time.

`->host_stop()` is called after all `->port_stop()` calls have completed. The hook must finalize hardware shutdown, release DMA and other resources, etc. This hook may be specified as NULL, in which case it is not called.

33.3 Error handling

This chapter describes how errors are handled under libata. Readers are advised to read SCSI EH (Documentation/scsi/scsi_eh.rst) and ATA exceptions doc first.

33.3.1 Origins of commands

In libata, a command is represented with `struct ata_queued_cmd` or qc. qc's are preallocated during port initialization and repetitively used for command executions. Currently only one qc is allocated per port but yet-to-be-merged NCQ branch allocates one for each tag and maps each qc to NCQ tag 1-to-1.

libata commands can originate from two sources - libata itself and SCSI midlayer. libata internal commands are used for initialization and error handling. All normal blk requests and commands for SCSI emulation are passed as SCSI commands through queuecommand callback of SCSI host template.

33.3.2 How commands are issued

Internal commands First, qc is allocated and initialized using `ata_qc_new_init()`. Although `ata_qc_new_init()` doesn't implement any wait or retry mechanism when qc is not available, internal commands are currently issued only during initialization and error recovery, so no other command is active and allocation is guaranteed to succeed.

Once allocated qc's taskfile is initialized for the command to be executed. qc currently has two mechanisms to notify completion. One is via `qc->complete_fn()` callback and the other is completion `qc->waiting`. `qc->complete_fn()` callback is the asynchronous path used by normal SCSI translated commands and `qc->waiting` is the synchronous (issuer sleeps in process context) path used by internal commands.

Once initialization is complete, host_set lock is acquired and the qc is issued.

SCSI commands All libata drivers use `ata_scsi_queuecmd()` as `hostt->queuecommand` callback. scmds can either be simulated or translated. No qc is involved in processing a simulated scmd. The result is computed right away and the scmd is completed.

For a translated scmd, `ata_qc_new_init()` is invoked to allocate a qc and the scmd is translated into the qc. SCSI midlayer's completion notification function pointer is stored into `qc->scsidone`.

`qc->complete_fn()` callback is used for completion notification. ATA commands use `ata_scsi_qc_complete()` while ATAPI commands use `atapi_qc_complete()`. Both functions end up calling `qc->scsidone` to notify upper layer when the qc is finished. After translation is completed, the qc is issued with `ata_qc_issue()`.

Note that SCSI midlayer invokes hostt->queuecommand while holding host_set lock, so all above occur while holding host_set lock.

33.3.3 How commands are processed

Depending on which protocol and which controller are used, commands are processed differently. For the purpose of discussion, a controller which uses taskfile interface and all standard callbacks is assumed.

Currently 6 ATA command protocols are used. They can be sorted into the following four categories according to how they are processed.

ATA NO DATA or DMA ATA_PROT_NODATA and ATA_PROT_DMA fall into this category. These types of commands don't require any software intervention once issued. Device will raise interrupt on completion.

ATA PIO ATA_PROT_PIO is in this category. libata currently implements PIO with polling. ATA_NIEN bit is set to turn off interrupt and pio_task on ata_wq performs polling and IO.

ATAPI NODATA or DMA ATA_PROT_ATAPI_NODATA and ATA_PROT_ATAPI_DMA are in this category. packet_task is used to poll BSY bit after issuing PACKET command. Once BSY is turned off by the device, packet_task transfers CDB and hands off processing to interrupt handler.

ATAPI PIO ATA_PROT_ATAPI is in this category. ATA_NIEN bit is set and, as in ATAPI NODATA or DMA, packet_task submits cdb. However, after submitting cdb, further processing (data transfer) is handed off to pio_task.

33.3.4 How commands are completed

Once issued, all qc's are either completed with `ata_qc_complete()` or time out. For commands which are handled by interrupts, `ata_host_intr()` invokes `ata_qc_complete()`, and, for PIO tasks, pio_task invokes `ata_qc_complete()`. In error cases, packet_task may also complete commands.

`ata_qc_complete()` does the following.

1. DMA memory is unmapped.
2. ATA_QCFLAG_ACTIVE is cleared from qc->flags.
3. `qc->complete_fn()` callback is invoked. If the return value of the callback is not zero. Completion is short circuited and `ata_qc_complete()` returns.

4. `__ata_qc_complete()` is called, which does
    1. `qc->flags` is cleared to zero.
    2. `ap->active_tag` and `qc->tag` are poisoned.
    3. `qc->waiting` is cleared & completed (in that order).
    4. qc is deallocated by clearing appropriate bit in `ap->qactive`.

So, it basically notifies upper layer and deallocates qc. One exception is short-circuit path in #3 which is used by `atapi_qc_complete()`.

For all non-ATAPI commands, whether it fails or not, almost the same code path is taken and very little error handling takes place. A qc is completed with success status if it succeeded, with failed status otherwise.

However, failed ATAPI commands require more handling as REQUEST SENSE is needed to acquire sense data. If an ATAPI command fails, `ata_qc_complete()` is invoked with error status, which in turn invokes `atapi_qc_complete()` via `qc->complete_fn()` callback.

This makes `atapi_qc_complete()` set `scmd->result` to SAM_STAT_CHECK_CONDITION, complete the scmd and return 1. As the sense data is empty but `scmd->result` is CHECK CONDITION, SCSI midlayer will invoke EH for the scmd, and returning 1 makes `ata_qc_complete()` to return without deallocating the qc. This leads us to `ata_scsi_error()` with partially completed qc.

33.3.5 ata_scsi_error()

`ata_scsi_error()` is the current `transportt->eh_strategy_handler()` for libata. As discussed above, this will be entered in two cases - timeout and ATAPI error completion. This function calls low level libata driver's `eng_timeout()` callback, the standard callback for which is `ata_eng_timeout()`. It checks if a qc is active and calls `ata_qc_timeout()` on the qc if so. Actual error handling occurs in `ata_qc_timeout()`.

If EH is invoked for timeout, `ata_qc_timeout()` stops BMDMA and completes the qc. Note that as we're currently in EH, we cannot call scsi_done. As described in SCSI EH doc, a recovered scmd should be either retried with `scsi_queue_insert()` or finished with `scsi_finish_command()`. Here, we override `qc->scsidone` with `scsi_finish_command()` and calls `ata_qc_complete()`.

If EH is invoked due to a failed ATAPI qc, the qc here is completed but not deallocated. The purpose of this half-completion is to use the qc as place holder to make EH code reach this place. This is a bit hackish, but it works.

Once control reaches here, the qc is deallocated by invoking `__ata_qc_complete()` explicitly. Then, internal qc for REQUEST SENSE is issued. Once sense data is acquired, scmd is finished by directly invoking `scsi_finish_command()` on the scmd. Note that as we already have completed and deallocated the qc which was associated with the scmd, we don't need to/cannot call `ata_qc_complete()` again.

33.3.6 Problems with the current EH

- Error representation is too crude. Currently any and all error conditions are represented with ATA STATUS and ERROR registers. Errors which aren't ATA device errors are treated as ATA device errors by setting ATA_ERR bit. Better error descriptor which can properly represent ATA and other errors/exceptions is needed.
- When handling timeouts, no action is taken to make device forget about the timed out command and ready for new commands.
- EH handling via `ata_scsi_error()` is not properly protected from usual command processing. On EH entrance, the device is not in quiescent state. Timed out commands may succeed or fail any time. pio_task and atapi_task may still be running.
- Too weak error recovery. Devices / controllers causing HSM mismatch errors and other errors quite often require reset to return to known state. Also, advanced error handling is necessary to support features like NCQ and hotplug.
- ATA errors are directly handled in the interrupt handler and PIO errors in pio_task. This is problematic for advanced error handling for the following reasons.

  First, advanced error handling often requires context and internal qc execution.

  Second, even a simple failure (say, CRC error) needs information gathering and could trigger complex error handling (say, resetting & reconfiguring). Having multiple code paths to gather information, enter EH and trigger actions makes life painful.

  Third, scattered EH code makes implementing low level drivers difficult. Low level drivers override libata callbacks. If EH is scattered over several places, each affected callbacks should perform its part of error handling. This can be error prone and painful.

33.4 libata Library

struct ata_link * **ata_link_next**(struct ata_link * link, struct ata_port * ap,
enum ata_link_iter_mode mode)

link iteration helper

Parameters

struct ata_link * link the previous link, NULL to start

struct ata_port * ap ATA port containing links to iterate

enum ata_link_iter_mode mode iteration mode, one of ATA_LITER_*

LOCKING: Host lock or EH context.

Return

Pointer to the next link.

struct ata_device * **ata_dev_next**(struct ata_device * dev, struct ata_link * link, enum ata_dev_iter_mode mode)
device iteration helper

Parameters

struct ata_device * dev the previous device, NULL to start

struct ata_link * link ATA link containing devices to iterate

enum ata_dev_iter_mode mode iteration mode, one of ATA_DITER_*

LOCKING: Host lock or EH context.

Return

Pointer to the next device.

int **atapi_cmd_type**(u8 opcode)
Determine ATAPI command type from SCSI opcode

Parameters

u8 opcode SCSI opcode

Determine ATAPI command type from **opcode**.

LOCKING: None.

Return

ATAPI_{READ|WRITE|READ_CD|PASS_THRU|MISC}

unsigned long **ata_pack_xfermask**(unsigned long pio_mask, unsigned long mwdma_mask, unsigned long udma_mask)
Pack pio, mwdma and udma masks into xfer_mask

Parameters

unsigned long pio_mask pio_mask

unsigned long mwdma_mask mwdma_mask

unsigned long udma_mask udma_mask

Pack **pio_mask**, **mwdma_mask** and **udma_mask** into a single unsigned int xfer_mask.

LOCKING: None.

Return

Packed xfer_mask.

u8 **ata_xfer_mask2mode**(unsigned long xfer_mask)
Find matching XFER_* for the given xfer_mask

Parameters

unsigned long xfer_mask xfer_mask of interest

Return matching XFER_* value for **xfer_mask**. Only the highest bit of **xfer_mask** is considered.

LOCKING: None.

Return

Matching XFER_* value, 0xff if no match found.

unsigned long **ata_xfer_mode2mask**(u8 xfer_mode)
Find matching xfer_mask for XFER_*

Parameters

u8 xfer_mode XFER_* of interest

Return matching xfer_mask for **xfer_mode**.

LOCKING: None.

Return

Matching xfer_mask, 0 if no match found.

int **ata_xfer_mode2shift**(unsigned long xfer_mode)
Find matching xfer_shift for XFER_*

Parameters

unsigned long xfer_mode XFER_* of interest

Return matching xfer_shift for **xfer_mode**.

LOCKING: None.

Return

Matching xfer_shift, -1 if no match found.

const char * **ata_mode_string**(unsigned long xfer_mask)
convert xfer_mask to string

Parameters

unsigned long xfer_mask mask of bits supported; only highest bit counts.

Determine string which represents the highest speed (highest bit in **mode_mask**).

LOCKING: None.

Return

Constant C string representing highest speed listed in **mode_mask**, or the constant C string “<n/a>”.

unsigned int **ata_dev_classify**(const struct ata_taskfile * tf)
determine device type based on ATA-spec signature

Parameters

const struct ata_taskfile * tf ATA taskfile register set for device to be identified

Determine from taskfile register contents whether a device is ATA or ATAPI, as per “Signature and persistence” section of ATA/PI spec (volume 1, sect 5.14).

LOCKING: None.

Return

Device type, `ATA_DEV_ATA`, `ATA_DEV_ATAPI`, `ATA_DEV_PMP`, `ATA_DEV_ZAC`, or `ATA_DEV_UNKNOWN` the event of failure.

void **ata_id_string**(const u16 * id, unsigned char * s, unsigned int ofs, unsigned int len)
Convert IDENTIFY DEVICE page into string

Parameters

const u16 * id IDENTIFY DEVICE results we will examine

unsigned char * s string into which data is output

unsigned int ofs offset into identify device page

unsigned int len length of string to return. must be an even number.

The strings in the IDENTIFY DEVICE page are broken up into 16-bit chunks. Run through the string, and output each 8-bit chunk linearly, regardless of platform.

LOCKING: caller.

void **ata_id_c_string**(const u16 * id, unsigned char * s, unsigned int ofs, unsigned int len)
Convert IDENTIFY DEVICE page into C string

Parameters

const u16 * id IDENTIFY DEVICE results we will examine

unsigned char * s string into which data is output

unsigned int ofs offset into identify device page

unsigned int len length of string to return. must be an odd number.

This function is identical to ata_id_string except that it trims trailing spaces and terminates the resulting string with null. **len** must be actual maximum length (even number) + 1.

LOCKING: caller.

unsigned long **ata_id_xfermask**(const u16 * id)
Compute xfermask from the given IDENTIFY data

Parameters

const u16 * id IDENTIFY data to compute xfer mask from

Compute the xfermask for this device. This is not as trivial as it seems if we must consider early devices correctly.

FIXME: pre IDE drive timing (do we care ?).

LOCKING: None.

Return

Computed xfermask

unsigned int **ata_pio_need_iordy**(const struct ata_device * adev)
check if iordy needed

Parameters

const struct ata_device * adev ATA device

Check if the current speed of the device requires IORDY. Used by various controllers for chip configuration.

unsigned int **ata_do_dev_read_id**(struct ata_device * dev, struct ata_taskfile * tf, u16 * id)
default ID read method

Parameters

struct ata_device * dev device

struct ata_taskfile * tf proposed taskfile

u16 * id data buffer

Issue the identify taskfile and hand back the buffer containing identify data. For some RAID controllers and for pre ATA devices this function is wrapped or replaced by the driver

int **ata_cable_40wire**(struct ata_port * ap)
return 40 wire cable type

Parameters

struct ata_port * ap port

Helper method for drivers which want to hardwire 40 wire cable detection.

int **ata_cable_80wire**(struct ata_port * ap)
return 80 wire cable type

Parameters

struct ata_port * ap port

Helper method for drivers which want to hardwire 80 wire cable detection.

int **ata_cable_unknown**(struct ata_port * ap)
return unknown PATA cable.

Parameters

struct ata_port * ap port

Helper method for drivers which have no PATA cable detection.

int **ata_cable_ignore**(struct ata_port * ap)
return ignored PATA cable.

Parameters

struct ata_port * ap port

Helper method for drivers which don't use cable type to limit transfer mode.

int **ata_cable_sata**(struct ata_port * ap)
return SATA cable type

Parameters

struct ata_port * ap port

Helper method for drivers which have SATA cables

struct ata_device * **ata_dev_pair**(struct ata_device * adev)
return other device on cable

Parameters

struct ata_device * adev device

Obtain the other device on the same cable, or if none is present NULL is returned

int **ata_do_set_mode**(struct ata_link * link, struct ata_device ** r_failed_dev)
Program timings and issue SET FEATURES - XFER

Parameters

struct ata_link * link link on which timings will be programmed

struct ata_device ** r_failed_dev out parameter for failed device

Standard implementation of the function used to tune and set ATA device disk transfer mode (PIO3, UDMA6, etc.). If ata_dev_set_mode() fails, pointer to the failing device is returned in **r_failed_dev**.

LOCKING: PCI/etc. bus probe sem.

Return

0 on success, negative errno otherwise

int **ata_wait_after_reset**(struct ata_link * link, unsigned long deadline, int (*check_ready)(struct ata_link *link))
wait for link to become ready after reset

Parameters

struct ata_link * link link to be waited on

unsigned long deadline deadline jiffies for the operation

int (*)(struct ata_link *link) check_ready callback to check link readiness

Wait for **link** to become ready after reset.

LOCKING: EH context.

Return

0 if **link** is ready before **deadline**; otherwise, -errno.

int **ata_std_prereset**(struct ata_link * link, unsigned long deadline)
prepare for reset

Parameters

struct ata_link * link ATA link to be reset

unsigned long deadline deadline jiffies for the operation

link is about to be reset. Initialize it. Failure from prereset makes libata abort whole reset sequence and give up that port, so prereset should be best-effort. It does its best to prepare for reset sequence but if things go wrong, it should just whine, not fail.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno otherwise.

int **sata_std_hardreset**(struct ata_link * link, unsigned int * class, unsigned long deadline)
COMRESET w/o waiting or classification

Parameters

struct ata_link * link link to reset

unsigned int * class resulting class of attached device

unsigned long deadline deadline jiffies for the operation

Standard SATA COMRESET w/o waiting or classification.

LOCKING: Kernel thread context (may sleep)

Return

0 if link offline, -EAGAIN if link online, -errno on errors.

void **ata_std_postreset**(struct ata_link * link, unsigned int * classes)
standard postreset callback

Parameters

struct ata_link * link the target ata_link

unsigned int * classes classes of attached devices

This function is invoked after a successful reset. Note that the device might have been reset more than once using different reset methods before postreset is invoked.

LOCKING: Kernel thread context (may sleep)

unsigned int **ata_dev_set_feature**(struct ata_device * dev, u8 enable, u8 feature)
Issue SET FEATURES - SATA FEATURES

Parameters

struct ata_device * dev Device to which command will be sent

u8 enable Whether to enable or disable the feature

u8 feature The sector count represents the feature to set

Issue SET FEATURES - SATA FEATURES command to device **dev** on port **ap** with sector count

LOCKING: PCI/etc. bus probe sem.

Return

0 on success, AC_ERR_* mask otherwise.

int **ata_std_qc_defer**(struct ata_queued_cmd * qc)
Check whether a qc needs to be deferred

Parameters

struct ata_queued_cmd * qc ATA command in question

Non-NCQ commands cannot run with any other command, NCQ or not. As upper layer only knows the queue depth, we are responsible for maintaining exclusion. This function checks whether a new command **qc** can be issued.

LOCKING: spin_lock_irqsave(host lock)

Return

ATA_DEFER_* if deferring is needed, 0 otherwise.

void **ata_qc_complete**(struct ata_queued_cmd * qc)
Complete an active ATA command

Parameters

struct ata_queued_cmd * qc Command to complete

Indicate to the mid and upper layers that an ATA command has completed, with either an ok or not-ok status.

Refrain from calling this function multiple times when successfully completing multiple NCQ commands. ata_qc_complete_multiple() should be used instead, which will properly update IRQ expect state.

LOCKING: spin_lock_irqsave(host lock)

u64 **ata_qc_get_active**(struct ata_port * ap)
get bitmask of active qcs

Parameters

struct ata_port * ap port in question

LOCKING: spin_lock_irqsave(host lock)

Return

Bitmask of active qcs

bool **ata_link_online**(struct ata_link * link)
test whether the given link is online

Parameters

struct ata_link * link ATA link to test

Test whether **link** is online. This is identical to `ata_phys_link_online()` when there's no slave link. When there's a slave link, this function should only be called on the master link and will return true if any of M/S links is online.

LOCKING: None.

Return

True if the port online status is available and online.

bool **ata_link_offline**(struct ata_link * link)
test whether the given link is offline

Parameters

struct ata_link * link ATA link to test

Test whether **link** is offline. This is identical to `ata_phys_link_offline()` when there's no slave link. When there's a slave link, this function should only be called on the master link and will return true if both M/S links are offline.

LOCKING: None.

Return

True if the port offline status is available and offline.

int **ata_host_suspend**(struct ata_host * host, pm_message_t mesg)
suspend host

Parameters

struct ata_host * host host to suspend

pm_message_t mesg PM message

Suspend **host**. Actual operation is performed by port suspend.

void **ata_host_resume**(struct ata_host * host)
resume host

Parameters

struct ata_host * host host to resume

Resume **host**. Actual operation is performed by port resume.

struct ata_host * **ata_host_alloc**(struct device * dev, int max_ports)
allocate and init basic ATA host resources

Parameters

struct device * dev generic device this host is associated with

int max_ports maximum number of ATA ports associated with this host

Allocate and initialize basic ATA host resources. LLD calls this function to allocate a host, initializes it fully and attaches it using `ata_host_register()`.

max_ports ports are allocated and host->n_ports is initialized to **max_ports**. The caller is allowed to decrease host->n_ports before calling `ata_host_register()`. The unused ports will be automatically freed on registration.

Return

Allocate ATA host on success, NULL on failure.

LOCKING: Inherited from calling layer (may sleep).

struct ata_host * **ata_host_alloc_pinfo**(struct device * dev, const struct ata_port_info *const * ppi, int n_ports)
alloc host and init with port_info array

Parameters

struct device * dev generic device this host is associated with

const struct ata_port_info *const * ppi array of ATA port_info to initialize host with

int n_ports number of ATA ports attached to this host

Allocate ATA host and initialize with info from **ppi**. If NULL terminated, **ppi** may contain fewer entries than **n_ports**. The last entry will be used for the remaining ports.

Return

Allocate ATA host on success, NULL on failure.

LOCKING: Inherited from calling layer (may sleep).

int **ata_host_start**(struct ata_host * host)
start and freeze ports of an ATA host

Parameters

struct ata_host * host ATA host to start ports for

Start and then freeze ports of **host**. Started status is recorded in host->flags, so this function can be called multiple times. Ports are guaranteed to get started only once. If host->ops isn't initialized yet, its set to the first non-dummy port ops.

LOCKING: Inherited from calling layer (may sleep).

Return

0 if all ports are started successfully, -errno otherwise.

void **ata_host_init**(struct ata_host * host, struct device * dev, struct ata_port_operations * ops)
Initialize a host struct for sas (ipr, libsas)

Parameters

struct ata_host * host host to initialize

struct device * dev device host is attached to

struct ata_port_operations * ops port_ops

int **ata_host_register**(struct ata_host * host, struct scsi_host_template * sht)
register initialized ATA host

Parameters

struct ata_host * host ATA host to register

struct scsi_host_template * sht template for SCSI host

Register initialized ATA host. **host** is allocated using `ata_host_alloc()` and fully initialized by LLD. This function starts ports, registers **host** with ATA and SCSI layers and probe registered devices.

LOCKING: Inherited from calling layer (may sleep).

Return

0 on success, -errno otherwise.

int **ata_host_activate**(struct ata_host * host, int irq, irq_handler_t irq_handler, unsigned long irq_flags, struct scsi_host_template * sht)
start host, request IRQ and register it

Parameters

struct ata_host * host target ATA host

int irq IRQ to request

irq_handler_t irq_handler irq_handler used when requesting IRQ

unsigned long irq_flags irq_flags used when requesting IRQ

struct scsi_host_template * sht scsi_host_template to use when registering the host

After allocating an ATA host and initializing it, most libata LLDs perform three steps to activate the host - start host, request IRQ and register it. This helper takes necessary arguments and performs the three steps in one go.

An invalid IRQ skips the IRQ registration and expects the host to have set polling mode on the port. In this case, **irq_handler** should be NULL.

LOCKING: Inherited from calling layer (may sleep).

Return

0 on success, -errno otherwise.

void **ata_host_detach**(struct ata_host * host)
Detach all ports of an ATA host

Parameters

struct ata_host * host Host to detach

Detach all ports of **host**.

LOCKING: Kernel thread context (may sleep).

void **ata_pci_remove_one**(struct pci_dev * pdev)
PCI layer callback for device removal

Parameters

struct pci_dev * pdev PCI device that was removed

PCI layer indicates to libata via this hook that hot-unplug or module unload event has occurred. Detach all ports. Resource release is handled via devres.

LOCKING: Inherited from PCI layer (may sleep).

int **ata_platform_remove_one**(struct platform_device * pdev)
Platform layer callback for device removal

Parameters

struct platform_device * pdev Platform device that was removed

Platform layer indicates to libata via this hook that hot-unplug or module unload event has occurred. Detach all ports. Resource release is handled via devres.

LOCKING: Inherited from platform layer (may sleep).

void **ata_msleep**(struct ata_port * ap, unsigned int msecs)
ATA EH owner aware msleep

Parameters

struct ata_port * ap ATA port to attribute the sleep to

unsigned int msecs duration to sleep in milliseconds

Sleeps **msecs**. If the current task is owner of **ap**'s EH, the ownership is released before going to sleep and reacquired after the sleep is complete. IOW, other ports sharing the **ap->host** will be allowed to own the EH while this task is sleeping.

LOCKING: Might sleep.

u32 **ata_wait_register**(struct ata_port * ap, void __iomem * reg, u32 mask, u32 val, unsigned long interval, unsigned long timeout)
wait until register value changes

Parameters

struct ata_port * ap ATA port to wait register for, can be NULL

void __iomem * reg IO-mapped register

u32 mask Mask to apply to read register value

u32 val Wait condition

unsigned long interval polling interval in milliseconds

unsigned long timeout timeout in milliseconds

Waiting for some bits of register to change is a common operation for ATA controllers. This function reads 32bit LE IO-mapped register **reg** and tests for the following condition.

(***reg** & mask) != val

If the condition is met, it returns; otherwise, the process is repeated after **interval_msec** until timeout.

LOCKING: Kernel thread context (may sleep)

Return

The final register value.

33.5 libata Core Internals

struct ata_link * **ata_dev_phys_link**(struct ata_device * dev)
find physical link for a device

Parameters

struct ata_device * dev ATA device to look up physical link for

Look up physical link which **dev** is attached to. Note that this is different from **dev->link** only when **dev** is on slave link. For all other cases, it's the same as **dev->link**.

LOCKING: Don't care.

Return

Pointer to the found physical link.

void **ata_force_cbl**(struct ata_port * ap)
force cable type according to libata.force

Parameters

struct ata_port * ap ATA port of interest

Force cable type according to libata.force and whine about it. The last entry which has matching port number is used, so it can be specified as part of device force parameters. For example, both "a:40c,1.00:udma4" and "1.00:40c,udma4" have the same effect.

LOCKING: EH context.

void **ata_force_link_limits**(struct ata_link * link)
force link limits according to libata.force

Parameters

struct ata_link * link ATA link of interest

Force link flags and SATA spd limit according to libata.force and whine about it. When only the port part is specified (e.g. 1:), the limit applies to all links connected to both the host link and all fan-out ports connected via PMP. If the device part is specified as 0 (e.g. 1.00:), it specifies the first fan-out link not the host link. Device number 15 always points to the host link whether PMP is attached or not. If the controller has slave link, device number 16 points to it.

LOCKING: EH context.

void **ata_force_xfermask**(struct ata_device * dev)
force xfermask according to libata.force

Parameters

struct ata_device * dev ATA device of interest

Force xfer_mask according to libata.force and whine about it. For consistency with link selection, device number 15 selects the first device connected to the host link.

LOCKING: EH context.

void **ata_force_horkage**(struct ata_device * dev)
force horkage according to libata.force

Parameters

struct ata_device * dev ATA device of interest

Force horkage according to libata.force and whine about it. For consistency with link selection, device number 15 selects the first device connected to the host link.

LOCKING: EH context.

int **ata_rwcmd_protocol**(struct ata_taskfile * tf, struct ata_device * dev)
set taskfile r/w commands and protocol

Parameters

struct ata_taskfile * tf command to examine and configure

struct ata_device * dev device tf belongs to

Examine the device configuration and tf->flags to calculate the proper read/write commands and protocol to use.

LOCKING: caller.

u64 **ata_tf_read_block**(const struct ata_taskfile * tf, struct ata_device * dev)
Read block address from ATA taskfile

Parameters

const struct ata_taskfile * tf ATA taskfile of interest

struct ata_device * dev ATA device **tf** belongs to

LOCKING: None.

Read block address from **tf**. This function can handle all three address formats - LBA, LBA48 and CHS. tf->protocol and flags select the address format to use.

Return

Block address read from **tf**.

int **ata_build_rw_tf**(struct ata_taskfile * tf, struct ata_device * dev, u64 block, u32 n_block, unsigned int tf_flags, unsigned int tag, int class)
Build ATA taskfile for given read/write request

Parameters

struct ata_taskfile * tf Target ATA taskfile

struct ata_device * dev ATA device **tf** belongs to

u64 block Block address

u32 n_block Number of blocks

unsigned int tf_flags RW/FUA etc…

unsigned int tag tag

int class IO priority class

LOCKING: None.

Build ATA taskfile **tf** for read/write request described by **block**, **n_block**, **tf_flags** and **tag** on **dev**.

Return

0 on success, -ERANGE if the request is too large for **dev**, -EINVAL if the request is invalid.

void **ata_unpack_xfermask**(unsigned long xfer_mask, unsigned long * pio_mask, unsigned long * mwdma_mask, unsigned long * udma_mask)

Unpack xfer_mask into pio, mwdma and udma masks

Parameters

unsigned long xfer_mask xfer_mask to unpack

unsigned long * pio_mask resulting pio_mask

unsigned long * mwdma_mask resulting mwdma_mask

unsigned long * udma_mask resulting udma_mask

Unpack **xfer_mask** into **pio_mask**, **mwdma_mask** and **udma_mask**. Any NULL destination masks will be ignored.

int **ata_read_native_max_address**(struct ata_device * dev, u64 * max_sectors)

Read native max address

Parameters

struct ata_device * dev target device

u64 * max_sectors out parameter for the result native max address

Perform an LBA48 or LBA28 native size query upon the device in question.

Return

0 on success, -EACCES if command is aborted by the drive. -EIO on other errors.

int **ata_set_max_sectors**(struct ata_device * dev, u64 new_sectors)

Set max sectors

Parameters

struct ata_device * dev target device

u64 new_sectors new max sectors value to set for the device

Set max sectors of **dev** to **new_sectors**.

Return

0 on success, -EACCES if command is aborted or denied (due to previous non-volatile SET_MAX) by the drive. -EIO on other errors.

int **ata_hpa_resize**(struct ata_device * dev)
Resize a device with an HPA set

Parameters

struct ata_device * dev Device to resize

Read the size of an LBA28 or LBA48 disk with HPA features and resize it if required to the full size of the media. The caller must check the drive has the HPA feature set enabled.

Return

0 on success, -errno on failure.

void **ata_dump_id**(const u16 * id)
IDENTIFY DEVICE info debugging output

Parameters

const u16 * id IDENTIFY DEVICE page to dump

Dump selected 16-bit words from the given IDENTIFY DEVICE page.

LOCKING: caller.

unsigned **ata_exec_internal_sg**(struct ata_device * dev, struct ata_taskfile * tf, const u8 * cdb, int dma_dir, struct scatterlist * sgl, unsigned int n_elem, unsigned long timeout)
execute libata internal command

Parameters

struct ata_device * dev Device to which the command is sent

struct ata_taskfile * tf Taskfile registers for the command and the result

const u8 * cdb CDB for packet command

int dma_dir Data transfer direction of the command

struct scatterlist * sgl sg list for the data buffer of the command

unsigned int n_elem Number of sg entries

unsigned long timeout Timeout in msecs (0 for default)

Executes libata internal command with timeout. **tf** contains command on entry and result on return. Timeout and error conditions are reported via return value. No recovery action is taken after a command times out. It's caller's duty to clean up after timeout.

LOCKING: None. Should be called with kernel context, might sleep.

Return

Zero on success, AC_ERR_* mask on failure

unsigned **ata_exec_internal**(struct ata_device * dev, struct ata_taskfile * tf, const u8 * cdb, int dma_dir, void * buf, unsigned int buflen, unsigned long timeout)
execute libata internal command

Parameters

struct ata_device * dev Device to which the command is sent

struct ata_taskfile * tf Taskfile registers for the command and the result

const u8 * cdb CDB for packet command

int dma_dir Data transfer direction of the command

void * buf Data buffer of the command

unsigned int buflen Length of data buffer

unsigned long timeout Timeout in msecs (0 for default)

Wrapper around `ata_exec_internal_sg()` which takes simple buffer instead of sg list.

LOCKING: None. Should be called with kernel context, might sleep.

Return

Zero on success, AC_ERR_* mask on failure

u32 **ata_pio_mask_no_iordy**(const struct ata_device * adev)
Return the non IORDY mask

Parameters

const struct ata_device * adev ATA device

Compute the highest mode possible if we are not using iordy. Return -1 if no iordy mode is available.

int **ata_dev_read_id**(struct ata_device * dev, unsigned int * p_class, unsigned int flags, u16 * id)
Read ID data from the specified device

Parameters

struct ata_device * dev target device

unsigned int * p_class pointer to class of the target device (may be changed)

unsigned int flags ATA_READID_* flags

u16 * id buffer to read IDENTIFY data into

Read ID data from the specified device. ATA_CMD_ID_ATA is performed on ATA devices and ATA_CMD_ID_ATAPI on ATAPI devices. This function also issues ATA_CMD_INIT_DEV_PARAMS for pre-ATA4 drives.

FIXME: ATA_CMD_ID_ATA is optional for early drives and right now we abort if we hit that case.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno otherwise.

unsigned int **ata_read_log_page**(struct ata_device * dev, u8 log, u8 page, void * buf, unsigned int sectors)
read a specific log page

Parameters

struct ata_device * dev target device

u8 log log to read

u8 page page to read

void * buf buffer to store read page

unsigned int sectors number of sectors to read

Read log page using READ_LOG_EXT command.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, AC_ERR_* mask otherwise.

int **ata_dev_configure**(struct ata_device * dev)
Configure the specified ATA/ATAPI device

Parameters

struct ata_device * dev Target device to configure

Configure **dev** according to **dev->id**. Generic and low-level driver specific fixups are also applied.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, -errno otherwise

int **ata_bus_probe**(struct ata_port * ap)
Reset and probe ATA bus

Parameters

struct ata_port * ap Bus to probe

Master ATA bus probing function. Initiates a hardware-dependent bus reset, then attempts to identify any devices found on the bus.

LOCKING: PCI/etc. bus probe sem.

Return

Zero on success, negative errno otherwise.

void **sata_print_link_status**(struct ata_link * link)
Print SATA link status

Parameters

struct ata_link * link SATA link to printk link status about

This function prints link speed and status of a SATA link.

LOCKING: None.

int **sata_down_spd_limit**(struct ata_link * link, u32 spd_limit)
adjust SATA spd limit downward

Parameters

struct ata_link * link Link to adjust SATA spd limit for

u32 spd_limit Additional limit

Adjust SATA spd limit of **link** downward. Note that this function only adjusts the limit. The change must be applied using sata_set_spd().

If **spd_limit** is non-zero, the speed is limited to equal to or lower than **spd_limit** if such speed is supported. If **spd_limit** is slower than any supported speed, only the lowest supported speed is allowed.

LOCKING: Inherited from caller.

Return

0 on success, negative errno on failure

u8 **ata_timing_cycle2mode**(unsigned int xfer_shift, int cycle)
find xfer mode for the specified cycle duration

Parameters

unsigned int xfer_shift ATA_SHIFT_* value for transfer type to examine.

int cycle cycle duration in ns

Return matching xfer mode for **cycle**. The returned mode is of the transfer type specified by **xfer_shift**. If **cycle** is too slow for **xfer_shift**, 0xff is returned. If **cycle** is faster than the fastest known mode, the fasted mode is returned.

LOCKING: None.

Return

Matching xfer_mode, 0xff if no match found.

int **ata_down_xfermask_limit**(struct ata_device * dev, unsigned int sel)
adjust dev xfer masks downward

Parameters

struct ata_device * dev Device to adjust xfer masks

unsigned int sel ATA_DNXFER_* selector

Adjust xfer masks of **dev** downward. Note that this function does not apply the change. Invoking `ata_set_mode()` afterwards will apply the limit.

LOCKING: Inherited from caller.

Return

0 on success, negative errno on failure

int **ata_wait_ready**(struct ata_link * link, unsigned long deadline, int (*check_ready)(struct ata_link *link))
wait for link to become ready

Parameters

struct ata_link * link link to be waited on

unsigned long deadline deadline jiffies for the operation

int (*)(struct ata_link *link) check_ready callback to check link readiness

Wait for **link** to become ready. **check_ready** should return positive number if **link** is ready, 0 if it isn't, -ENODEV if link doesn't seem to be occupied, other errno for other error conditions.

Transient -ENODEV conditions are allowed for ATA_TMOUT_FF_WAIT.

LOCKING: EH context.

Return

0 if **link** is ready before **deadline**; otherwise, -errno.

int **ata_dev_same_device**(struct ata_device * dev, unsigned int new_class, const u16 * new_id)
Determine whether new ID matches configured device

Parameters

struct ata_device * dev device to compare against

unsigned int new_class class of the new device

const u16 * new_id IDENTIFY page of the new device

Compare **new_class** and **new_id** against **dev** and determine whether **dev** is the device indicated by **new_class** and **new_id**.

LOCKING: None.

Return

1 if **dev** matches **new_class** and **new_id**, 0 otherwise.

int **ata_dev_reread_id**(struct ata_device * dev, unsigned int readid_flags)
Re-read IDENTIFY data

Parameters

struct ata_device * dev target ATA device

unsigned int readid_flags read ID flags

Re-read IDENTIFY page and make sure **dev** is still attached to the port.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, negative errno otherwise

int **ata_dev_revalidate**(struct ata_device * dev, unsigned int new_class, unsigned int readid_flags)
Revalidate ATA device

Parameters

struct ata_device * dev device to revalidate

unsigned int new_class new class code

unsigned int readid_flags read ID flags

Re-read IDENTIFY page, make sure **dev** is still attached to the port and reconfigure it according to the new IDENTIFY page.

LOCKING: Kernel thread context (may sleep)

Return

0 on success, negative errno otherwise

int **ata_is_40wire**(struct ata_device * dev)
check drive side detection

Parameters

struct ata_device * dev device

Perform drive side detection decoding, allowing for device vendors who can't follow the documentation.

int **cable_is_40wire**(struct ata_port * ap)
40/80/SATA decider

Parameters

struct ata_port * ap port to consider

This function encapsulates the policy for speed management in one place. At the moment we don't cache the result but there is a good case for setting ap->cbl to the result when we are called with unknown cables (and figuring out if it impacts hotplug at all).

Return 1 if the cable appears to be 40 wire.

void **ata_dev_xfermask**(struct ata_device * dev)
Compute supported xfermask of the given device

Parameters

struct ata_device * dev Device to compute xfermask for

Compute supported xfermask of **dev** and store it in dev->*_mask. This function is responsible for applying all known limits including host controller limits, device blacklist, etc…

LOCKING: None.

unsigned int **ata_dev_set_xfermode**(struct ata_device * dev)
Issue SET FEATURES - XFER MODE command

Parameters

struct ata_device * dev Device to which command will be sent

Issue SET FEATURES - XFER MODE command to device **dev** on port **ap**.

LOCKING: PCI/etc. bus probe sem.

Return

0 on success, AC_ERR_* mask otherwise.

unsigned int **ata_dev_init_params**(struct ata_device * dev, u16 heads, u16 sectors)
Issue INIT DEV PARAMS command

Parameters

struct ata_device * dev Device to which command will be sent

u16 heads Number of heads (taskfile parameter)

u16 sectors Number of sectors (taskfile parameter)

LOCKING: Kernel thread context (may sleep)

Return

0 on success, AC_ERR_* mask otherwise.

int **atapi_check_dma**(struct ata_queued_cmd * qc)
Check whether ATAPI DMA can be supported

Parameters

struct ata_queued_cmd * qc Metadata associated with taskfile to check

Allow low-level driver to filter ATA PACKET commands, returning a status indicating whether or not it is OK to use DMA for the supplied PACKET command.

LOCKING: spin_lock_irqsave(host lock)

Return

0 when ATAPI DMA can be used nonzero otherwise

void **ata_sg_init**(struct ata_queued_cmd * qc, struct scatterlist * sg, unsigned int n_elem)
Associate command with scatter-gather table.

Parameters

struct ata_queued_cmd * qc Command to be associated

struct scatterlist * sg Scatter-gather table.

unsigned int n_elem Number of elements in s/g table.

Initialize the data-related elements of queued_cmd **qc** to point to a scatter-gather table **sg**, containing **n_elem** elements.

LOCKING: spin_lock_irqsave(host lock)

void **ata_sg_clean**(struct ata_queued_cmd * qc)
Unmap DMA memory associated with command

Parameters

struct ata_queued_cmd * qc Command containing DMA memory to be released

Unmap all mapped DMA memory associated with this command.

LOCKING: spin_lock_irqsave(host lock)

int **ata_sg_setup**(struct ata_queued_cmd * qc)
DMA-map the scatter-gather table associated with a command.

Parameters

struct ata_queued_cmd * qc Command with scatter-gather table to be mapped.

DMA-map the scatter-gather table associated with queued_cmd **qc**.

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, negative on error.

void **swap_buf_le16**(u16 * buf, unsigned int buf_words)
swap halves of 16-bit words in place

Parameters

u16 * buf Buffer to swap

unsigned int buf_words Number of 16-bit words in buffer.

Swap halves of 16-bit words if needed to convert from little-endian byte order to native cpu byte order, or vice-versa.

LOCKING: Inherited from caller.

struct ata_queued_cmd * **ata_qc_new_init**(struct ata_device * dev, int tag)
Request an available ATA command, and initialize it

Parameters

struct ata_device * dev Device from whom we request an available command structure

int tag tag

LOCKING: None.

void **ata_qc_free**(struct ata_queued_cmd * qc)
free unused ata_queued_cmd

Parameters

struct ata_queued_cmd * qc Command to complete

Designed to free unused ata_queued_cmd object in case something prevents using it.

LOCKING: spin_lock_irqsave(host lock)

void **ata_qc_issue**(struct ata_queued_cmd * qc)
issue taskfile to device

Parameters

struct ata_queued_cmd * qc command to issue to device

Prepare an ATA command to submission to device. This includes mapping the data into a DMA-able area, filling in the S/G table, and finally writing the taskfile to hardware, starting the command.

LOCKING: spin_lock_irqsave(host lock)

bool **ata_phys_link_online**(struct ata_link * link)
test whether the given link is online

Parameters

struct ata_link * link ATA link to test

Test whether **link** is online. Note that this function returns 0 if online status of **link** cannot be obtained, so ata_link_online(link) != !ata_link_offline(link).

LOCKING: None.

Return

True if the port online status is available and online.

bool **ata_phys_link_offline**(struct ata_link * link)
test whether the given link is offline

Parameters

struct ata_link * link ATA link to test

Test whether **link** is offline. Note that this function returns 0 if offline status of **link** cannot be obtained, so ata_link_online(link) != !ata_link_offline(link).

LOCKING: None.

Return

True if the port offline status is available and offline.

void **ata_dev_init**(struct ata_device * dev)
Initialize an ata_device structure

Parameters

struct ata_device * dev Device structure to initialize

Initialize **dev** in preparation for probing.

LOCKING: Inherited from caller.

void **ata_link_init**(struct ata_port * ap, struct ata_link * link, int pmp)
Initialize an ata_link structure

Parameters

struct ata_port * ap ATA port link is attached to

struct ata_link * link Link structure to initialize

int pmp Port multiplier port number

Initialize **link**.

LOCKING: Kernel thread context (may sleep)

int **sata_link_init_spd**(struct ata_link * link)
Initialize link->sata_spd_limit

Parameters

struct ata_link * link Link to configure sata_spd_limit for

Initialize `link->[hw_]sata_spd_limit` to the currently configured value.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, -errno on failure.

struct ata_port * **ata_port_alloc**(struct ata_host * host)
allocate and initialize basic ATA port resources

Parameters

struct ata_host * host ATA host this allocated port belongs to

Allocate and initialize basic ATA port resources.

Return

Allocate ATA port on success, NULL on failure.

LOCKING: Inherited from calling layer (may sleep).

void **ata_finalize_port_ops**(struct ata_port_operations * ops)
finalize ata_port_operations

Parameters

struct ata_port_operations * ops ata_port_operations to finalize

An ata_port_operations can inherit from another ops and that ops can again inherit from another. This can go on as many times as necessary as long as there is no loop in the inheritance chain.

Ops tables are finalized when the host is started. NULL or unspecified entries are inherited from the closet ancestor which has the method and the entry is populated with it. After finalization, the ops table directly points to all the methods and ->inherits is no longer necessary and cleared.

Using ATA_OP_NULL, inheriting ops can force a method to NULL.

LOCKING: None.

void **ata_port_detach**(struct ata_port * ap)
Detach ATA port in preparation of device removal

Parameters

struct ata_port * ap ATA port to be detached

Detach all ATA devices and the associated SCSI devices of **ap**; then, remove the associated SCSI host. **ap** is guaranteed to be quiescent on return from this function.

LOCKING: Kernel thread context (may sleep).

void **__ata_ehi_push_desc**(struct ata_eh_info * ehi, const char * fmt, ...)
push error description without adding separator

Parameters

struct ata_eh_info * ehi target EHI

const char * fmt printf format string

Format string according to **fmt** and append it to **ehi->desc**.

LOCKING: spin_lock_irqsave(host lock)

... variable arguments

void **ata_ehi_push_desc**(struct ata_eh_info * ehi, const char * fmt, ...)
push error description with separator

Parameters

struct ata_eh_info * ehi target EHI

const char * fmt printf format string

Format string according to **fmt** and append it to **ehi->desc**. If **ehi->desc** is not empty, ", " is added in-between.

LOCKING: spin_lock_irqsave(host lock)

... variable arguments

void **ata_ehi_clear_desc**(struct ata_eh_info * ehi)
clean error description

Parameters

struct ata_eh_info * ehi target EHI

Clear **ehi->desc**.

LOCKING: spin_lock_irqsave(host lock)

void **ata_port_desc**(struct ata_port * ap, const char * fmt, ...)
append port description

Parameters

struct ata_port * ap target ATA port

const char * fmt printf format string

Format string according to **fmt** and append it to port description. If port description is not empty, "" is added in-between. This function is to be used while initializing ata_host. The description is printed on host registration.

LOCKING: None.

... variable arguments

void **ata_port_pbar_desc**(struct ata_port * ap, int bar, ssize_t offset, const char * name)
append PCI BAR description

Parameters

struct ata_port * ap target ATA port

int bar target PCI BAR

ssize_t offset offset into PCI BAR

const char * name name of the area

If **offset** is negative, this function formats a string which contains the name, address, size and type of the BAR and appends it to the port description. If **offset** is zero or positive, only name and offsetted address is appended.

LOCKING: None.

unsigned long **ata_internal_cmd_timeout**(struct ata_device * dev, u8 cmd)
determine timeout for an internal command

Parameters

struct ata_device * dev target device

u8 cmd internal command to be issued

Determine timeout for internal command **cmd** for **dev**.

LOCKING: EH context.

Return

Determined timeout.

void **ata_internal_cmd_timed_out**(struct ata_device * dev, u8 cmd)
notification for internal command timeout

Parameters

struct ata_device * dev target device

u8 cmd internal command which timed out

Notify EH that internal command **cmd** for **dev** timed out. This function should be called only for commands whose timeouts are determined using `ata_internal_cmd_timeout()`.

LOCKING: EH context.

void **ata_eh_acquire**(struct ata_port * ap)
acquire EH ownership

Parameters

struct ata_port * ap ATA port to acquire EH ownership for

Acquire EH ownership for **ap**. This is the basic exclusion mechanism for ports sharing a host. Only one port hanging off the same host can claim the ownership of EH.

LOCKING: EH context.

void **ata_eh_release**(struct ata_port * ap)
release EH ownership

Parameters

struct ata_port * ap ATA port to release EH ownership for

Release EH ownership for **ap** if the caller. The caller must have acquired EH ownership using `ata_eh_acquire()` previously.

LOCKING: EH context.

void **ata_scsi_error**(struct Scsi_Host * host)
SCSI layer error handler callback

Parameters

struct Scsi_Host * host SCSI host on which error occurred

Handles SCSI-layer-thrown error events.

LOCKING: Inherited from SCSI layer (none, can sleep)

Return

Zero.

void **ata_scsi_cmd_error_handler**(struct Scsi_Host * host, struct ata_port * ap, struct list_head * eh_work_q)
error callback for a list of commands

Parameters

struct Scsi_Host * host scsi host containing the port

struct ata_port * ap ATA port within the host

struct list_head * eh_work_q list of commands to process

Description

process the given list of commands and return those finished to the ap->eh_done_q. This function is the first part of the libata error handler which processes a given list of failed commands.

void **ata_scsi_port_error_handler**(struct Scsi_Host * host, struct ata_port * ap)
recover the port after the commands

Parameters

struct Scsi_Host * host SCSI host containing the port

struct ata_port * ap the ATA port

Description

Handle the recovery of the port **ap** after all the commands have been recovered.

void **ata_port_wait_eh**(struct ata_port * ap)
Wait for the currently pending EH to complete

Parameters

struct ata_port * ap Port to wait EH for

Wait until the currently pending EH is complete.

LOCKING: Kernel thread context (may sleep).

void **ata_eh_set_pending**(struct ata_port * ap, int fastdrain)
set ATA_PFLAG_EH_PENDING and activate fast drain

Parameters

struct ata_port * ap target ATA port

int fastdrain activate fast drain

Set ATA_PFLAG_EH_PENDING and activate fast drain if **fastdrain** is non-zero and EH wasn't pending before. Fast drain ensures that EH kicks in in timely manner.

LOCKING: spin_lock_irqsave(host lock)

void **ata_qc_schedule_eh**(struct ata_queued_cmd * qc)
schedule qc for error handling

Parameters

struct ata_queued_cmd * qc command to schedule error handling for

Schedule error handling for **qc**. EH will kick in as soon as other commands are drained.

LOCKING: spin_lock_irqsave(host lock)

void **ata_std_sched_eh**(struct ata_port * ap)
non-libsas ata_ports issue eh with this common routine

Parameters

struct ata_port * ap ATA port to schedule EH for

LOCKING: inherited from ata_port_schedule_eh spin_lock_irqsave(host lock)

void **ata_std_end_eh**(struct ata_port * ap)
non-libsas ata_ports complete eh with this common routine

Parameters

struct ata_port * ap ATA port to end EH for

Description

In the libata object model there is a 1:1 mapping of ata_port to shost, so host fields can be directly manipulated under ap->lock, in the libsas case we need to hold a lock at the ha->level to coordinate these events.

LOCKING: spin_lock_irqsave(host lock)

void **ata_port_schedule_eh**(struct ata_port * ap)
schedule error handling without a qc

Parameters

struct ata_port * ap ATA port to schedule EH for

Schedule error handling for **ap**. EH will kick in as soon as all commands are drained.

LOCKING: spin_lock_irqsave(host lock)

int **ata_link_abort**(struct ata_link * link)
abort all qc's on the link

Parameters

struct ata_link * link ATA link to abort qc's for

Abort all active qc's active on **link** and schedule EH.

LOCKING: spin_lock_irqsave(host lock)

Return

Number of aborted qc's.

int **ata_port_abort**(struct ata_port * ap)
abort all qc's on the port

Parameters

struct ata_port * ap ATA port to abort qc's for

Abort all active qc's of **ap** and schedule EH.

LOCKING: spin_lock_irqsave(host_set lock)

Return

Number of aborted qc's.

void **__ata_port_freeze**(struct ata_port * ap)
freeze port

Parameters

struct ata_port * ap ATA port to freeze

This function is called when HSM violation or some other condition disrupts normal operation of the port. Frozen port is not allowed to perform any operation until the port is thawed, which usually follows a successful reset.

ap->ops->freeze() callback can be used for freezing the port hardware-wise (e.g. mask interrupt and stop DMA engine). If a port cannot be frozen hardware-wise, the interrupt handler must ack and clear interrupts unconditionally while the port is frozen.

LOCKING: spin_lock_irqsave(host lock)

int **ata_port_freeze**(struct ata_port * ap)
abort & freeze port

Parameters

struct ata_port * ap ATA port to freeze

Abort and freeze **ap**. The freeze operation must be called first, because some hardware requires special operations before the taskfile registers are accessible.

LOCKING: spin_lock_irqsave(host lock)

Return

Number of aborted commands.

void **ata_eh_freeze_port**(struct ata_port * ap)
EH helper to freeze port

Parameters

struct ata_port * ap ATA port to freeze

Freeze **ap**.

LOCKING: None.

void **ata_eh_thaw_port**(struct ata_port * ap)
EH helper to thaw port

Parameters

struct ata_port * ap ATA port to thaw

Thaw frozen port **ap**.

LOCKING: None.

void **ata_eh_qc_complete**(struct ata_queued_cmd * qc)
Complete an active ATA command from EH

Parameters

struct ata_queued_cmd * qc Command to complete

Indicate to the mid and upper layers that an ATA command has completed. To be used from EH.

void **ata_eh_qc_retry**(struct ata_queued_cmd * qc)
Tell midlayer to retry an ATA command after EH

Parameters

struct ata_queued_cmd * qc Command to retry

Indicate to the mid and upper layers that an ATA command should be retried. To be used from EH.

SCSI midlayer limits the number of retries to scmd->allowed. scmd->allowed is incremented for commands which get retried due to unrelated failures (qc->err_mask is zero).

void **ata_dev_disable**(struct ata_device * dev)
disable ATA device

Parameters

struct ata_device * dev ATA device to disable

Disable **dev**.

Locking: EH context.

void **ata_eh_detach_dev**(struct ata_device * dev)
detach ATA device

Parameters

struct ata_device * dev ATA device to detach

Detach **dev**.

LOCKING: None.

void **ata_eh_about_to_do**(struct ata_link * link, struct ata_device * dev, unsigned int action)
about to perform eh_action

Parameters

struct ata_link * link target ATA link

struct ata_device * dev target ATA dev for per-dev action (can be NULL)

unsigned int action action about to be performed

Called just before performing EH actions to clear related bits in **link->eh_info** such that eh actions are not unnecessarily repeated.

LOCKING: None.

void **ata_eh_done**(struct ata_link * link, struct ata_device * dev, unsigned int action)
EH action complete

Parameters

struct ata_link * link ATA link for which EH actions are complete

struct ata_device * dev target ATA dev for per-dev action (can be NULL)

unsigned int action action just completed

Called right after performing EH actions to clear related bits in **link->eh_context**.

LOCKING: None.

const char * **ata_err_string**(unsigned int err_mask)
convert err_mask to descriptive string

Parameters

unsigned int err_mask error mask to convert to string

Convert **err_mask** to descriptive string. Errors are prioritized according to severity and only the most severe error is reported.

LOCKING: None.

Return

Descriptive string for **err_mask**

unsigned int **atapi_eh_tur**(struct ata_device * dev, u8 * r_sense_key)
perform ATAPI TEST_UNIT_READY

Parameters

struct ata_device * dev target ATAPI device

u8 * r_sense_key out parameter for sense_key

Perform ATAPI TEST_UNIT_READY.

LOCKING: EH context (may sleep).

Return

0 on success, AC_ERR_* mask on failure.

void **ata_eh_request_sense**(struct ata_queued_cmd * qc, struct scsi_cmnd * cmd)
perform REQUEST_SENSE_DATA_EXT

Parameters

struct ata_queued_cmd * qc qc to perform REQUEST_SENSE_SENSE_DATA_EXT to

struct scsi_cmnd * cmd scsi command for which the sense code should be set

Perform REQUEST_SENSE_DATA_EXT after the device reported CHECK SENSE. This function is an EH helper.

LOCKING: Kernel thread context (may sleep).

unsigned int **atapi_eh_request_sense**(struct ata_device * dev, u8 * sense_buf, u8 dfl_sense_key)
perform ATAPI REQUEST_SENSE

Parameters

struct ata_device * dev device to perform REQUEST_SENSE to

u8 * sense_buf result sense data buffer (SCSI_SENSE_BUFFERSIZE bytes long)

u8 dfl_sense_key default sense key to use

Perform ATAPI REQUEST_SENSE after the device reported CHECK SENSE. This function is EH helper.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, AC_ERR_* mask on failure

void **ata_eh_analyze_serror**(struct ata_link * link)
analyze SError for a failed port

Parameters

struct ata_link * link ATA link to analyze SError for

Analyze SError if available and further determine cause of failure.

LOCKING: None.

unsigned int **ata_eh_analyze_tf**(struct ata_queued_cmd * qc, const struct ata_taskfile * tf)
analyze taskfile of a failed qc

Parameters

struct ata_queued_cmd * qc qc to analyze

const struct ata_taskfile * tf Taskfile registers to analyze

Analyze taskfile of **qc** and further determine cause of failure. This function also requests ATAPI sense data if available.

LOCKING: Kernel thread context (may sleep).

Return

Determined recovery action

unsigned int **ata_eh_speed_down_verdict**(struct ata_device * dev)
Determine speed down verdict

Parameters

struct ata_device * dev Device of interest

This function examines error ring of **dev** and determines whether NCQ needs to be turned off, transfer speed should be stepped down, or falling back to PIO is necessary.

ECAT_ATA_BUS : ATA_BUS error for any command

ECAT_TOUT_HSM [TIMEOUT for any command or HSM violation for] IO commands

ECAT_UNK_DEV : Unknown DEV error for IO commands

ECAT_DUBIOUS_* [Identical to above three but occurred while] data transfer hasn't been verified.

Verdicts are

NCQ_OFF : Turn off NCQ.

SPEED_DOWN [Speed down transfer speed but don't fall back] to PIO.

FALLBACK_TO_PIO : Fall back to PIO.

Even if multiple verdicts are returned, only one action is taken per error. An action triggered by non-DUBIOUS errors clears ering, while one triggered by DUBIOUS_* errors doesn't. This is to expedite speed down decisions right after device is initially configured.

The following are speed down rules. #1 and #2 deal with DUBIOUS errors.

1. If more than one DUBIOUS_ATA_BUS or DUBIOUS_TOUT_HSM errors occurred during last 5 mins, SPEED_DOWN and FALLBACK_TO_PIO.
2. If more than one DUBIOUS_TOUT_HSM or DUBIOUS_UNK_DEV errors occurred during last 5 mins, NCQ_OFF.
3. If more than 8 ATA_BUS, TOUT_HSM or UNK_DEV errors occurred during last 5 mins, FALLBACK_TO_PIO
4. If more than 3 TOUT_HSM or UNK_DEV errors occurred during last 10 mins, NCQ_OFF.
5. If more than 3 ATA_BUS or TOUT_HSM errors, or more than 6 UNK_DEV errors occurred during last 10 mins, SPEED_DOWN.

LOCKING: Inherited from caller.

Return

OR of ATA_EH_SPDN_* flags.

unsigned int **ata_eh_speed_down**(struct ata_device * dev, unsigned int eflags, unsigned int err_mask)

record error and speed down if necessary

Parameters

struct ata_device * dev Failed device

unsigned int eflags mask of ATA_EFLAG_* flags

unsigned int err_mask err_mask of the error

Record error and examine error history to determine whether adjusting transmission speed is necessary. It also sets transmission limits appropriately if such adjustment is necessary.

LOCKING: Kernel thread context (may sleep).

Return

Determined recovery action.

int **ata_eh_worth_retry**(struct ata_queued_cmd * qc)
analyze error and decide whether to retry

Parameters

struct ata_queued_cmd * qc qc to possibly retry

Look at the cause of the error and decide if a retry might be useful or not. We don't want to retry media errors because the drive itself has probably already taken 10-30 seconds doing its own internal retries before reporting the failure.

bool **ata_eh_quiet**(struct ata_queued_cmd * qc)
check if we need to be quiet about a command error

Parameters

struct ata_queued_cmd * qc qc to check

Look at the qc flags anbd its scsi command request flags to determine if we need to be quiet about the command failure.

void **ata_eh_link_autopsy**(struct ata_link * link)
analyze error and determine recovery action

Parameters

struct ata_link * link host link to perform autopsy on

Analyze why **link** failed and determine which recovery actions are needed. This function also sets more detailed AC_ERR_* values and fills sense data for ATAPI CHECK SENSE.

LOCKING: Kernel thread context (may sleep).

void **ata_eh_autopsy**(struct ata_port * ap)
analyze error and determine recovery action

Parameters

struct ata_port * ap host port to perform autopsy on

Analyze all links of **ap** and determine why they failed and which recovery actions are needed.

LOCKING: Kernel thread context (may sleep).

const char * **ata_get_cmd_descript**(u8 command)
get description for ATA command

Parameters

u8 command ATA command code to get description for

Return a textual description of the given command, or NULL if the command is not known.

LOCKING: None

void **ata_eh_link_report**(struct ata_link * link)
report error handling to user

Parameters

struct ata_link * link ATA link EH is going on

Report EH to user.

LOCKING: None.

void **ata_eh_report**(struct ata_port * ap)
report error handling to user

Parameters

struct ata_port * ap ATA port to report EH about

Report EH to user.

LOCKING: None.

int **ata_set_mode**(struct ata_link * link, struct ata_device ** r_failed_dev)
Program timings and issue SET FEATURES - XFER

Parameters

struct ata_link * link link on which timings will be programmed

struct ata_device ** r_failed_dev out parameter for failed device

Set ATA device disk transfer mode (PIO3, UDMA6, etc.). If `ata_set_mode()` fails, pointer to the failing device is returned in **r_failed_dev**.

LOCKING: PCI/etc. bus probe sem.

Return

0 on success, negative errno otherwise

int **atapi_eh_clear_ua**(struct ata_device * dev)
Clear ATAPI UNIT ATTENTION after reset

Parameters

struct ata_device * dev ATAPI device to clear UA for

Resets and other operations can make an ATAPI device raise UNIT ATTENTION which causes the next operation to fail. This function clears UA.

LOCKING: EH context (may sleep).

Return

0 on success, -errno on failure.

int **ata_eh_maybe_retry_flush**(struct ata_device * dev)
Retry FLUSH if necessary

Parameters

struct ata_device * dev ATA device which may need FLUSH retry

If **dev** failed FLUSH, it needs to be reported upper layer immediately as it means that **dev** failed to remap and already lost at least a sector and further FLUSH retrials won't make any difference to the lost sector. However, if FLUSH failed for other reasons, for example transmission error, FLUSH needs to be retried.

This function determines whether FLUSH failure retry is necessary and performs it if so.

Return

0 if EH can continue, -errno if EH needs to be repeated.

int **ata_eh_set_lpm**(struct ata_link * link, enum ata_lpm_policy policy, struct ata_device ** r_failed_dev)
configure SATA interface power management

Parameters

struct ata_link * link link to configure power management

enum ata_lpm_policy policy the link power management policy

struct ata_device ** r_failed_dev out parameter for failed device

Enable SATA Interface power management. This will enable Device Interface Power Management (DIPM) for min_power and medium_power_with_dipm policies, and then call driver specific callbacks for enabling Host Initiated Power management.

LOCKING: EH context.

Return

0 on success, -errno on failure.

int **ata_eh_recover**(struct ata_port * ap, ata_prereset_fn_t prereset, ata_reset_fn_t softreset, ata_reset_fn_t hardreset, ata_postreset_fn_t postreset, struct ata_link ** r_failed_link)
recover host port after error

Parameters

struct ata_port * ap host port to recover

ata_prereset_fn_t prereset prereset method (can be NULL)

ata_reset_fn_t softreset softreset method (can be NULL)

ata_reset_fn_t hardreset hardreset method (can be NULL)

ata_postreset_fn_t postreset postreset method (can be NULL)

struct ata_link ** r_failed_link out parameter for failed link

This is the alpha and omega, eum and yang, heart and soul of libata exception handling. On entry, actions required to recover each link and hotplug requests are recorded in the link's eh_context. This function executes all the

operations with appropriate retrials and fallbacks to resurrect failed devices, detach goners and greet newcomers.

LOCKING: Kernel thread context (may sleep).

Return

0 on success, -errno on failure.

void **ata_eh_finish**(struct ata_port * ap)
finish up EH

Parameters

struct ata_port * ap host port to finish EH for

Recovery is complete. Clean up EH states and retry or finish failed qcs.

LOCKING: None.

void **ata_do_eh**(struct ata_port * ap, ata_prereset_fn_t prereset, ata_reset_fn_t softreset, ata_reset_fn_t hardreset, ata_postreset_fn_t postreset)
do standard error handling

Parameters

struct ata_port * ap host port to handle error for

ata_prereset_fn_t prereset prereset method (can be NULL)

ata_reset_fn_t softreset softreset method (can be NULL)

ata_reset_fn_t hardreset hardreset method (can be NULL)

ata_postreset_fn_t postreset postreset method (can be NULL)

Perform standard error handling sequence.

LOCKING: Kernel thread context (may sleep).

void **ata_std_error_handler**(struct ata_port * ap)
standard error handler

Parameters

struct ata_port * ap host port to handle error for

Standard error handler

LOCKING: Kernel thread context (may sleep).

void **ata_eh_handle_port_suspend**(struct ata_port * ap)
perform port suspend operation

Parameters

struct ata_port * ap port to suspend

Suspend **ap**.

LOCKING: Kernel thread context (may sleep).

void **ata_eh_handle_port_resume**(struct ata_port * ap)
perform port resume operation

Parameters

struct ata_port * ap port to resume

Resume **ap**.

LOCKING: Kernel thread context (may sleep).

33.6 libata SCSI translation/emulation

int **ata_std_bios_param**(struct scsi_device * sdev, struct block_device * bdev, sector_t capacity, int geom)
generic bios head/sector/cylinder calculator used by sd.

Parameters

struct scsi_device * sdev SCSI device for which BIOS geometry is to be determined

struct block_device * bdev block device associated with **sdev**

sector_t capacity capacity of SCSI device

int geom location to which geometry will be output

Generic bios head/sector/cylinder calculator used by sd. Most BIOSes nowadays expect a XXX/255/16 (CHS) mapping. Some situations may arise where the disk is not bootable if this is not used.

LOCKING: Defined by the SCSI layer. We don't really care.

Return

Zero.

void **ata_scsi_unlock_native_capacity**(struct scsi_device * sdev)
unlock native capacity

Parameters

struct scsi_device * sdev SCSI device to adjust device capacity for

This function is called if a partition on **sdev** extends beyond the end of the device. It requests EH to unlock HPA.

LOCKING: Defined by the SCSI layer. Might sleep.

bool **ata_scsi_dma_need_drain**(struct request * rq)
Check whether data transfer may overflow

Parameters

struct request * rq request to be checked

ATAPI commands which transfer variable length data to host might overflow due to application error or hardware bug. This function checks whether overflow should be drained and ignored for **request**.

LOCKING: None.

Return

1 if ; otherwise, 0.

int **ata_scsi_slave_config**(struct scsi_device * sdev)
Set SCSI device attributes

Parameters

struct scsi_device * sdev SCSI device to examine

This is called before we actually start reading and writing to the device, to configure certain SCSI mid-layer behaviors.

LOCKING: Defined by SCSI layer. We don't really care.

void **ata_scsi_slave_destroy**(struct scsi_device * sdev)
SCSI device is about to be destroyed

Parameters

struct scsi_device * sdev SCSI device to be destroyed

sdev is about to be destroyed for hot/warm unplugging. If this unplugging was initiated by libata as indicated by NULL dev->sdev, this function doesn't have to do anything. Otherwise, SCSI layer initiated warm-unplug is in progress. Clear dev->sdev, schedule the device for ATA detach and invoke EH.

LOCKING: Defined by SCSI layer. We don't really care.

int **ata_scsi_queuecmd**(struct Scsi_Host * shost, struct scsi_cmnd * cmd)
Issue SCSI cdb to libata-managed device

Parameters

struct Scsi_Host * shost SCSI host of command to be sent

struct scsi_cmnd * cmd SCSI command to be sent

In some cases, this function translates SCSI commands into ATA taskfiles, and queues the taskfiles to be sent to hardware. In other cases, this function simulates a SCSI device by evaluating and responding to certain SCSI commands. This creates the overall effect of ATA and ATAPI devices appearing as SCSI devices.

LOCKING: ATA host lock

Return

Return value from __ata_scsi_queuecmd() if **cmd** can be queued, 0 otherwise.

int **ata_get_identity**(struct ata_port * ap, struct scsi_device * sdev, void __user * arg)
Handler for HDIO_GET_IDENTITY ioctl

Parameters

struct ata_port * ap target port

struct scsi_device * sdev SCSI device to get identify data for

void __user * arg User buffer area for identify data

LOCKING: Defined by the SCSI layer. We don’t really care.

Return

Zero on success, negative errno on error.

int **ata_cmd_ioctl**(struct scsi_device * scsidev, void __user * arg)
Handler for HDIO_DRIVE_CMD ioctl

Parameters

struct scsi_device * scsidev Device to which we are issuing command

void __user * arg User provided data for issuing command

LOCKING: Defined by the SCSI layer. We don’t really care.

Return

Zero on success, negative errno on error.

int **ata_task_ioctl**(struct scsi_device * scsidev, void __user * arg)
Handler for HDIO_DRIVE_TASK ioctl

Parameters

struct scsi_device * scsidev Device to which we are issuing command

void __user * arg User provided data for issuing command

LOCKING: Defined by the SCSI layer. We don’t really care.

Return

Zero on success, negative errno on error.

struct ata_queued_cmd * **ata_scsi_qc_new**(struct ata_device * dev, struct scsi_cmnd * cmd)
acquire new ata_queued_cmd reference

Parameters

struct ata_device * dev ATA device to which the new command is attached

struct scsi_cmnd * cmd SCSI command that originated this ATA command

Obtain a reference to an unused ata_queued_cmd structure, which is the basic libata structure representing a single ATA command sent to the hardware.

If a command was available, fill in the SCSI-specific portions of the structure with information on the current command.

LOCKING: spin_lock_irqsave(host lock)

Return

Command allocated, or `NULL` if none available.

void **ata_dump_status**(unsigned id, struct ata_taskfile * tf)
user friendly display of error info

Parameters

unsigned id id of the port in question

struct ata_taskfile * tf ptr to filled out taskfile

Decode and dump the ATA error/status registers for the user so that they have some idea what really happened at the non make-believe layer.

LOCKING: inherited from caller

void **ata_to_sense_error**(unsigned id, u8 drv_stat, u8 drv_err, u8 * sk, u8 * asc, u8 * ascq, int verbose)
convert ATA error to SCSI error

Parameters

unsigned id ATA device number

u8 drv_stat value contained in ATA status register

u8 drv_err value contained in ATA error register

u8 * sk the sense key we'll fill out

u8 * asc the additional sense code we'll fill out

u8 * ascq the additional sense code qualifier we'll fill out

int verbose be verbose

Converts an ATA error into a SCSI error. Fill out pointers to SK, ASC, and ASCQ bytes for later use in fixed or descriptor format sense blocks.

LOCKING: spin_lock_irqsave(host lock)

void **ata_gen_ata_sense**(struct ata_queued_cmd * qc)
generate a SCSI fixed sense block

Parameters

struct ata_queued_cmd * qc Command that we are erroring out

Generate sense block for a failed ATA command **qc**. Descriptor format is used to accommodate LBA48 block address.

LOCKING: None.

unsigned int **ata_scsi_start_stop_xlat**(struct ata_queued_cmd * qc)
Translate SCSI START STOP UNIT command

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

Sets up an ATA taskfile to issue STANDBY (to stop) or READ VERIFY (to start). Perhaps these commands should be preceded by CHECK POWER MODE to see what power mode the device is already in. [See SAT revision 5 at www.t10.org]

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on error.

unsigned int **ata_scsi_flush_xlat**(struct ata_queued_cmd * qc)
Translate SCSI SYNCHRONIZE CACHE command

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

Sets up an ATA taskfile to issue FLUSH CACHE or FLUSH CACHE EXT.

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on error.

void **scsi_6_lba_len**(const u8 * cdb, u64 * plba, u32 * plen)
Get LBA and transfer length

Parameters

const u8 * cdb SCSI command to translate

Calculate LBA and transfer length for 6-byte commands.

u64 * plba the LBA

u32 * plen the transfer length

void **scsi_10_lba_len**(const u8 * cdb, u64 * plba, u32 * plen)
Get LBA and transfer length

Parameters

const u8 * cdb SCSI command to translate

Calculate LBA and transfer length for 10-byte commands.

u64 * plba the LBA

u32 * plen the transfer length

void **scsi_16_lba_len**(const u8 * cdb, u64 * plba, u32 * plen)
Get LBA and transfer length

Parameters

const u8 * cdb SCSI command to translate

Calculate LBA and transfer length for 16-byte commands.

u64 * plba the LBA

u32 * plen the transfer length

unsigned int **ata_scsi_verify_xlat**(struct ata_queued_cmd * qc)
Translate SCSI VERIFY command into an ATA one

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

Converts SCSI VERIFY command to an ATA READ VERIFY command.

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on error.

unsigned int **ata_scsi_rw_xlat**(struct ata_queued_cmd * qc)
Translate SCSI r/w command into an ATA one

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

Converts any of six SCSI read/write commands into the ATA counterpart, including starting sector (LBA), sector count, and taking into account the device's LBA48 support.

Commands `READ_6`, `READ_10`, `READ_16`, `WRITE_6`, `WRITE_10`, and `WRITE_16` are currently supported.

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on error.

int **ata_scsi_translate**(struct ata_device * dev, struct scsi_cmnd * cmd, ata_xlat_func_t xlat_func)
Translate then issue SCSI command to ATA device

Parameters

struct ata_device * dev ATA device to which the command is addressed

struct scsi_cmnd * cmd SCSI command to execute

ata_xlat_func_t xlat_func Actor which translates **cmd** to an ATA taskfile

Our ->queuecommand() function has decided that the SCSI command issued can be directly translated into an ATA command, rather than handled internally.

This function sets up an ata_queued_cmd structure for the SCSI command, and sends that ata_queued_cmd to the hardware.

The xlat_func argument (actor) returns 0 if ready to execute ATA command, else 1 to finish translation. If 1 is returned then cmd->result (and possibly cmd->sense_buffer) are assumed to be set reflecting an error condition or clean (early) termination.

LOCKING: spin_lock_irqsave(host lock)

Return

0 on success, SCSI_ML_QUEUE_DEVICE_BUSY if the command needs to be deferred.

void * **ata_scsi_rbuf_get**(struct scsi_cmnd * cmd, bool copy_in, unsigned long * flags)
Map response buffer.

Parameters

struct scsi_cmnd * cmd SCSI command containing buffer to be mapped.

bool copy_in copy in from user buffer

Prepare buffer for simulated SCSI commands.

LOCKING: spin_lock_irqsave(ata_scsi_rbuf_lock) on success

unsigned long * flags unsigned long variable to store irq enable status

Return

Pointer to response buffer.

void **ata_scsi_rbuf_put**(struct scsi_cmnd * cmd, bool copy_out, unsigned long * flags)

Unmap response buffer.

Parameters

struct scsi_cmnd * cmd SCSI command containing buffer to be unmapped.

bool copy_out copy out result

unsigned long * flags flags passed to `ata_scsi_rbuf_get()`

Returns rbuf buffer. The result is copied to **cmd**'s buffer if **copy_back** is true.

LOCKING: Unlocks ata_scsi_rbuf_lock.

void **ata_scsi_rbuf_fill**(struct ata_scsi_args * args, unsigned int (*actor)(struct ata_scsi_args *args, u8 *rbuf))

wrapper for SCSI command simulators

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

unsigned int (*)(struct ata_scsi_args *args, u8 *rbuf) actor Callback hook for desired SCSI command simulator

Takes care of the hard work of simulating a SCSI command···Mapping the response buffer, calling the command's handler, and handling the handler's return value. This return value indicates whether the handler wishes the SCSI command to be completed successfully (0), or not (in which case cmd->result and sense buffer are assumed to be set).

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_std**(struct ata_scsi_args * args, u8 * rbuf)

Simulate INQUIRY command

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Returns standard device identification data associated with non-VPD INQUIRY command output.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_00**(struct ata_scsi_args * args, u8 * rbuf)

Simulate INQUIRY VPD page 0, list of pages

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Returns list of inquiry VPD pages available.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_80**(struct ata_scsi_args * args, u8 * rbuf)
Simulate INQUIRY VPD page 80, device serial number

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Returns ATA device serial number.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_83**(struct ata_scsi_args * args, u8 * rbuf)
Simulate INQUIRY VPD page 83, device identity

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Yields two logical unit device identification designators:

- vendor specific ASCII containing the ATA serial number
- SAT defined "t10 vendor id based" containing ASCII vendor name ("ATA "), model and serial numbers.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_inq_89**(struct ata_scsi_args * args, u8 * rbuf)
Simulate INQUIRY VPD page 89, ATA info

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Yields SAT-specified ATA VPD page.

LOCKING: spin_lock_irqsave(host lock)

void **modecpy**(u8 * dest, const u8 * src, int n, bool changeable)
Prepare response for MODE SENSE

Parameters

u8 * dest output buffer

const u8 * src data being copied

int n length of mode page

bool changeable whether changeable parameters are requested

Generate a generic MODE SENSE page for either current or changeable parameters.

LOCKING: None.

unsigned int **ata_msense_caching**(u16 * id, u8 * buf, bool changeable)
Simulate MODE SENSE caching info page

Parameters

u16 * id device IDENTIFY data

u8 * buf output buffer

bool changeable whether changeable parameters are requested

Generate a caching info page, which conditionally indicates write caching to the SCSI layer, depending on device capabilities.

LOCKING: None.

unsigned int **ata_msense_control**(struct ata_device * dev, u8 * buf, bool changeable)
Simulate MODE SENSE control mode page

Parameters

struct ata_device * dev ATA device of interest

u8 * buf output buffer

bool changeable whether changeable parameters are requested

Generate a generic MODE SENSE control mode page.

LOCKING: None.

unsigned int **ata_msense_rw_recovery**(u8 * buf, bool changeable)
Simulate MODE SENSE r/w error recovery page

Parameters

u8 * buf output buffer

bool changeable whether changeable parameters are requested

Generate a generic MODE SENSE r/w error recovery page.

LOCKING: None.

unsigned int **ata_scsiop_mode_sense**(struct ata_scsi_args * args, u8 * rbuf)
Simulate MODE SENSE 6, 10 commands

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Simulate MODE SENSE commands. Assume this is invoked for direct access devices (e.g. disks) only. There should be no block descriptor for other device types.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsiop_read_cap**(struct ata_scsi_args * args, u8 * rbuf)
Simulate READ CAPACITY[16] commands

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Simulate READ CAPACITY commands.

LOCKING: None.

unsigned int **ata_scsiop_report_luns**(struct ata_scsi_args * args, u8 * rbuf)
Simulate REPORT LUNS command

Parameters

struct ata_scsi_args * args device IDENTIFY data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Simulate REPORT LUNS command.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **atapi_xlat**(struct ata_queued_cmd * qc)
Initialize PACKET taskfile

Parameters

struct ata_queued_cmd * qc command structure to be initialized

LOCKING: spin_lock_irqsave(host lock)

Return

Zero on success, non-zero on failure.

struct ata_device * **ata_scsi_find_dev**(struct ata_port * ap, const struct scsi_device * scsidev)
lookup ata_device from scsi_cmnd

Parameters

struct ata_port * ap ATA port to which the device is attached

const struct scsi_device * scsidev SCSI device from which we derive the ATA device

Given various information provided in struct scsi_cmnd, map that onto an ATA bus, and using that mapping determine which ata_device is associated with the SCSI command to be sent.

LOCKING: spin_lock_irqsave(host lock)

Return

Associated ATA device, or NULL if not found.

unsigned int **ata_scsi_pass_thru**(struct ata_queued_cmd * qc)
convert ATA pass-thru CDB to taskfile

Parameters

struct ata_queued_cmd * qc command structure to be initialized

Handles either 12, 16, or 32-byte versions of the CDB.

Return

Zero on success, non-zero on failure.

size_t **ata_format_dsm_trim_descr**(struct scsi_cmnd * cmd, u32 trmax, u64 sector, u32 count)
SATL Write Same to DSM Trim

Parameters

struct scsi_cmnd * cmd SCSI command being translated

u32 trmax Maximum number of entries that will fit in sector_size bytes.

u64 sector Starting sector

u32 count Total Range of request in logical sectors

Description

Rewrite the WRITE SAME descriptor to be a DSM TRIM little-endian formatted descriptor.

Upto 64 entries of the format:

63:48 Range Length 47:0 LBA

Range Length of 0 is ignored. LBA's should be sorted order and not overlap.

NOTE

this is the same format as ADD LBA(S) TO NV CACHE PINNED SET

Return

Number of bytes copied into sglist.

unsigned int **ata_scsi_write_same_xlat**(struct ata_queued_cmd * qc)
SATL Write Same to ATA SCT Write Same

Parameters

struct ata_queued_cmd * qc Command to be translated

Description

Translate a SCSI WRITE SAME command to be either a DSM TRIM command or an SCT Write Same command. Based on WRITE SAME has the UNMAP flag:

- When set translate to DSM TRIM

- When clear translate to SCT Write Same

unsigned int **ata_scsiop_maint_in**(struct ata_scsi_args * args, u8 * rbuf)
Simulate a subset of MAINTENANCE_IN

Parameters

struct ata_scsi_args * args device MAINTENANCE_IN data / SCSI command of interest.

u8 * rbuf Response buffer, to which simulated SCSI cmd output is sent.

Yields a subset to satisfy `scsi_report_opcode()`

LOCKING: spin_lock_irqsave(host lock)

void **ata_scsi_report_zones_complete**(struct ata_queued_cmd * qc)
convert ATA output

Parameters

struct ata_queued_cmd * qc command structure returning the data

Convert T-13 little-endian field representation into T-10 big-endian field representation. What a mess.

int **ata_mselect_caching**(struct ata_queued_cmd * qc, const u8 * buf, int len, u16 * fp)
Simulate MODE SELECT for caching info page

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

const u8 * buf input buffer

int len number of valid bytes in the input buffer

u16 * fp out parameter for the failed field on error

Prepare a taskfile to modify caching information for the device.

LOCKING: None.

int **ata_mselect_control**(struct ata_queued_cmd * qc, const u8 * buf, int len, u16 * fp)
Simulate MODE SELECT for control page

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

const u8 * buf input buffer

int len number of valid bytes in the input buffer

u16 * fp out parameter for the failed field on error

Prepare a taskfile to modify caching information for the device.

LOCKING: None.

unsigned int **ata_scsi_mode_select_xlat**(struct ata_queued_cmd * qc)
Simulate MODE SELECT 6, 10 commands

Parameters

struct ata_queued_cmd * qc Storage for translated ATA taskfile

Converts a MODE SELECT command to an ATA SET FEATURES taskfile. Assume this is invoked for direct access devices (e.g. disks) only. There should be no block descriptor for other device types.

LOCKING: spin_lock_irqsave(host lock)

unsigned int **ata_scsi_var_len_cdb_xlat**(struct ata_queued_cmd * qc)
SATL variable length CDB to Handler

Parameters

struct ata_queued_cmd * qc Command to be translated

Translate a SCSI variable length CDB to specified commands. It checks a service action value in CDB to call corresponding handler.

Return

Zero on success, non-zero on failure

ata_xlat_func_t **ata_get_xlat_func**(struct ata_device * dev, u8 cmd)
check if SCSI to ATA translation is possible

Parameters

struct ata_device * dev ATA device

u8 cmd SCSI command opcode to consider

Look up the SCSI command given, and determine whether the SCSI command is to be translated or simulated.

Return

Pointer to translation function if possible, `NULL` if not.

void **ata_scsi_dump_cdb**(struct ata_port * ap, struct scsi_cmnd * cmd)
dump SCSI command contents to dmesg

Parameters

struct ata_port * ap ATA port to which the command was being sent

struct scsi_cmnd * cmd SCSI command to dump

Prints the contents of a SCSI command via `printk()`.

void **ata_scsi_simulate**(struct ata_device * dev, struct scsi_cmnd * cmd)
simulate SCSI command on ATA device

Parameters

struct ata_device * dev the target device

struct scsi_cmnd * cmd SCSI command being sent to device.

Interprets and directly executes a select list of SCSI commands that can be handled internally.

LOCKING: spin_lock_irqsave(host lock)

int **ata_scsi_offline_dev**(struct ata_device * dev)
offline attached SCSI device

Parameters

struct ata_device * dev ATA device to offline attached SCSI device for

This function is called from ata_eh_hotplug() and responsible for taking the SCSI device attached to **dev** offline. This function is called with host lock which protects dev->sdev against clearing.

LOCKING: spin_lock_irqsave(host lock)

Return

1 if attached SCSI device exists, 0 otherwise.

void **ata_scsi_remove_dev**(struct ata_device * dev)
remove attached SCSI device

Parameters

struct ata_device * dev ATA device to remove attached SCSI device for

This function is called from ata_eh_scsi_hotplug() and responsible for removing the SCSI device attached to **dev**.

LOCKING: Kernel thread context (may sleep).

void **ata_scsi_media_change_notify**(struct ata_device * dev)
send media change event

Parameters

struct ata_device * dev Pointer to the disk device with media change event

Tell the block layer to send a media change notification event.

LOCKING: spin_lock_irqsave(host lock)

void **ata_scsi_hotplug**(struct work_struct * work)
SCSI part of hotplug

Parameters

struct work_struct * work Pointer to ATA port to perform SCSI hotplug on

Perform SCSI part of hotplug. It' s executed from a separate workqueue after EH completes. This is necessary because SCSI hot plugging requires working EH and hot unplugging is synchronized with hot plugging with a mutex.

LOCKING: Kernel thread context (may sleep).

int **ata_scsi_user_scan**(struct Scsi_Host * shost, unsigned int channel, unsigned int id, u64 lun)
indication for user-initiated bus scan

Parameters

struct Scsi_Host * shost SCSI host to scan

unsigned int channel Channel to scan

unsigned int id ID to scan

u64 lun LUN to scan

This function is called when user explicitly requests bus scan. Set probe pending flag and invoke EH.

LOCKING: SCSI layer (we don't care)

Return

Zero.

void **ata_scsi_dev_rescan**(struct work_struct * work)
initiate scsi_rescan_device()

Parameters

struct work_struct * work Pointer to ATA port to perform scsi_rescan_device()

After ATA pass thru (SAT) commands are executed successfully, libata need to propagate the changes to SCSI layer.

LOCKING: Kernel thread context (may sleep).

33.7 ATA errors and exceptions

This chapter tries to identify what error/exception conditions exist for ATA/ATAPI devices and describe how they should be handled in implementation-neutral way.

The term 'error' is used to describe conditions where either an explicit error condition is reported from device or a command has timed out.

The term 'exception' is either used to describe exceptional conditions which are not errors (say, power or hotplug events), or to describe both errors and non-error exceptional conditions. Where explicit distinction between error and exception is necessary, the term 'non-error exception' is used.

33.7.1 Exception categories

Exceptions are described primarily with respect to legacy taskfile + bus master IDE interface. If a controller provides other better mechanism for error reporting, mapping those into categories described below shouldn't be difficult.

In the following sections, two recovery actions - reset and reconfiguring transport - are mentioned. These are described further in EH recovery actions.

HSM violation

This error is indicated when STATUS value doesn't match HSM requirement during issuing or execution any ATA/ATAPI command.

- ATA_STATUS doesn't contain !BSY && DRDY && !DRQ while trying to issue a command.
- !BSY && !DRQ during PIO data transfer.
- DRQ on command completion.

- !BSY && ERR after CDB transfer starts but before the last byte of CDB is transferred. ATA/ATAPI standard states that "The device shall not terminate the PACKET command with an error before the last byte of the command packet has been written"in the error outputs description of PACKET command and the state diagram doesn't include such transitions.

In these cases, HSM is violated and not much information regarding the error can be acquired from STATUS or ERROR register. IOW, this error can be anything - driver bug, faulty device, controller and/or cable.

As HSM is violated, reset is necessary to restore known state. Reconfiguring transport for lower speed might be helpful too as transmission errors sometimes cause this kind of errors.

ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION)

These are errors detected and reported by ATA/ATAPI devices indicating device problems. For this type of errors, STATUS and ERROR register values are valid and describe error condition. Note that some of ATA bus errors are detected by ATA/ATAPI devices and reported using the same mechanism as device errors. Those cases are described later in this section.

For ATA commands, this type of errors are indicated by !BSY && ERR during command execution and on completion.

For ATAPI commands,

- !BSY && ERR && ABRT right after issuing PACKET indicates that PACKET command is not supported and falls in this category.
- !BSY && ERR(==CHK) && !ABRT after the last byte of CDB is transferred indicates CHECK CONDITION and doesn't fall in this category.
- !BSY && ERR(==CHK) && ABRT after the last byte of CDB is transferred *probably* indicates CHECK CONDITION and doesn't fall in this category.

Of errors detected as above, the following are not ATA/ATAPI device errors but ATA bus errors and should be handled according to ATA bus error.

CRC error during data transfer This is indicated by ICRC bit in the ERROR register and means that corruption occurred during data transfer. Up to ATA/ATAPI-7, the standard specifies that this bit is only applicable to UDMA transfers but ATA/ATAPI-8 draft revision 1f says that the bit may be applicable to multiword DMA and PIO.

ABRT error during data transfer or on completion Up to ATA/ATAPI-7, the standard specifies that ABRT could be set on ICRC errors and on cases where a device is not able to complete a command. Combined with the fact that MWDMA and PIO transfer errors aren't allowed to use ICRC bit up to ATA/ATAPI-7, it seems to imply that ABRT bit alone could indicate transfer errors.

However, ATA/ATAPI-8 draft revision 1f removes the part that ICRC errors can turn on ABRT. So, this is kind of gray area. Some heuristics are needed here.

ATA/ATAPI device errors can be further categorized as follows.

Media errors This is indicated by UNC bit in the ERROR register. ATA devices reports UNC error only after certain number of retries cannot recover the data, so there's nothing much else to do other than notifying upper layer.

READ and WRITE commands report CHS or LBA of the first failed sector but ATA/ATAPI standard specifies that the amount of transferred data on error completion is indeterminate, so we cannot assume that sectors preceding the failed sector have been transferred and thus cannot complete those sectors successfully as SCSI does.

Media changed / media change requested error <<TODO: fill here>>

Address error This is indicated by IDNF bit in the ERROR register. Report to upper layer.

Other errors This can be invalid command or parameter indicated by ABRT ERROR bit or some other error condition. Note that ABRT bit can indicate a lot of things including ICRC and Address errors. Heuristics needed.

Depending on commands, not all STATUS/ERROR bits are applicable. These non-applicable bits are marked with "na"in the output descriptions but up to ATA/ATAPI-7 no definition of "na" can be found. However, ATA/ATAPI-8 draft revision 1f describes "N/A" as follows.

> **3.2.3.3a N/A** A keyword the indicates a field has no defined value in this standard and should not be checked by the host or device. N/A fields should be cleared to zero.

So, it seems reasonable to assume that "na" bits are cleared to zero by devices and thus need no explicit masking.

ATAPI device CHECK CONDITION

ATAPI device CHECK CONDITION error is indicated by set CHK bit (ERR bit) in the STATUS register after the last byte of CDB is transferred for a PACKET command. For this kind of errors, sense data should be acquired to gather information regarding the errors. REQUEST SENSE packet command should be used to acquire sense data.

Once sense data is acquired, this type of errors can be handled similarly to other SCSI errors. Note that sense data may indicate ATA bus error (e.g. Sense Key 04h HARDWARE ERROR && ASC/ASCQ 47h/00h SCSI PARITY ERROR). In such cases, the error should be considered as an ATA bus error and handled according to ATA bus error.

ATA device error (NCQ)

NCQ command error is indicated by cleared BSY and set ERR bit during NCQ command phase (one or more NCQ commands outstanding). Although STATUS and ERROR registers will contain valid values describing the error, READ LOG EXT is required to clear the error condition, determine which command has failed and acquire more information.

READ LOG EXT Log Page 10h reports which tag has failed and taskfile register values describing the error. With this information the failed command can be handled as a normal ATA command error as in ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION) and all other in-flight commands must be retried. Note that this retry should not be counted - it's likely that commands retried this way would have completed normally if it were not for the failed command.

Note that ATA bus errors can be reported as ATA device NCQ errors. This should be handled as described in ATA bus error.

If READ LOG EXT Log Page 10h fails or reports NQ, we're thoroughly screwed. This condition should be treated according to HSM violation.

ATA bus error

ATA bus error means that data corruption occurred during transmission over ATA bus (SATA or PATA). This type of errors can be indicated by

- ICRC or ABRT error as described in ATA/ATAPI device error (non-NCQ / non-CHECK CONDITION).
- Controller-specific error completion with error information indicating transmission error.
- On some controllers, command timeout. In this case, there may be a mechanism to determine that the timeout is due to transmission error.
- Unknown/random errors, timeouts and all sorts of weirdities.

As described above, transmission errors can cause wide variety of symptoms ranging from device ICRC error to random device lockup, and, for many cases, there is no way to tell if an error condition is due to transmission error or not; therefore, it's necessary to employ some kind of heuristic when dealing with errors and timeouts. For example, encountering repetitive ABRT errors for known supported command is likely to indicate ATA bus error.

Once it's determined that ATA bus errors have possibly occurred, lowering ATA bus transmission speed is one of actions which may alleviate the problem. See Reconfigure transport for more information.

PCI bus error

Data corruption or other failures during transmission over PCI (or other system bus). For standard BMDMA, this is indicated by Error bit in the BMDMA Status register. This type of errors must be logged as it indicates something is very wrong with the system. Resetting host controller is recommended.

Late completion

This occurs when timeout occurs and the timeout handler finds out that the timed out command has completed successfully or with error. This is usually caused by lost interrupts. This type of errors must be logged. Resetting host controller is recommended.

Unknown error (timeout)

This is when timeout occurs and the command is still processing or the host and device are in unknown state. When this occurs, HSM could be in any valid or invalid state. To bring the device to known state and make it forget about the timed out command, resetting is necessary. The timed out command may be retried.

Timeouts can also be caused by transmission errors. Refer to ATA bus error for more details.

Hotplug and power management exceptions

<<TODO: fill here>>

33.7.2 EH recovery actions

This section discusses several important recovery actions.

Clearing error condition

Many controllers require its error registers to be cleared by error handler. Different controllers may have different requirements.

For SATA, it' s strongly recommended to clear at least SError register during error handling.

Reset

During EH, resetting is necessary in the following cases.

- HSM is in unknown or invalid state
- HBA is in unknown or invalid state
- EH needs to make HBA/device forget about in-flight commands
- HBA/device behaves weirdly

Resetting during EH might be a good idea regardless of error condition to improve EH robustness. Whether to reset both or either one of HBA and device depends on situation but the following scheme is recommended.

- When it's known that HBA is in ready state but ATA/ATAPI device is in unknown state, reset only device.
- If HBA is in unknown state, reset both HBA and device.

HBA resetting is implementation specific. For a controller complying to taskfile/BMDMA PCI IDE, stopping active DMA transaction may be sufficient iff BMDMA state is the only HBA context. But even mostly taskfile/BMDMA PCI IDE complying controllers may have implementation specific requirements and mechanism to reset themselves. This must be addressed by specific drivers.

OTOH, ATA/ATAPI standard describes in detail ways to reset ATA/ATAPI devices.

PATA hardware reset This is hardware initiated device reset signalled with asserted PATA RESET- signal. There is no standard way to initiate hardware reset from software although some hardware provides registers that allow driver to directly tweak the RESET- signal.

Software reset This is achieved by turning CONTROL SRST bit on for at least 5us. Both PATA and SATA support it but, in case of SATA, this may require controller-specific support as the second Register FIS to clear SRST should be transmitted while BSY bit is still set. Note that on PATA, this resets both master and slave devices on a channel.

EXECUTE DEVICE DIAGNOSTIC command Although ATA/ATAPI standard doesn't describe exactly, EDD implies some level of resetting, possibly similar level with software reset. Host-side EDD protocol can be handled with normal command processing and most SATA controllers should be able to handle EDD's just like other commands. As in software reset, EDD affects both devices on a PATA bus.

Although EDD does reset devices, this doesn't suit error handling as EDD cannot be issued while BSY is set and it's unclear how it will act when device is in unknown/weird state.

ATAPI DEVICE RESET command This is very similar to software reset except that reset can be restricted to the selected device without affecting the other device sharing the cable.

SATA phy reset This is the preferred way of resetting a SATA device. In effect, it's identical to PATA hardware reset. Note that this can be done with the standard SCR Control register. As such, it's usually easier to implement than software reset.

One more thing to consider when resetting devices is that resetting clears certain configuration parameters and they need to be set to their previous or newly adjusted values after reset.

Parameters affected are.

- CHS set up with INITIALIZE DEVICE PARAMETERS (seldom used)
- Parameters set with SET FEATURES including transfer mode setting
- Block count set with SET MULTIPLE MODE
- Other parameters (SET MAX, MEDIA LOCK…)

ATA/ATAPI standard specifies that some parameters must be maintained across hardware or software reset, but doesn't strictly specify all of them. Always reconfiguring needed parameters after reset is required for robustness. Note that this also applies when resuming from deep sleep (power-off).

Also, ATA/ATAPI standard requires that IDENTIFY DEVICE / IDENTIFY PACKET DEVICE is issued after any configuration parameter is updated or a hardware reset and the result used for further operation. OS driver is required to implement revalidation mechanism to support this.

Reconfigure transport

For both PATA and SATA, a lot of corners are cut for cheap connectors, cables or controllers and it's quite common to see high transmission error rate. This can be mitigated by lowering transmission speed.

The following is a possible scheme Jeff Garzik suggested.

> If more than $N (3?) transmission errors happen in 15 minutes,
>
> - if SATA, decrease SATA PHY speed. if speed cannot be decreased,
> - decrease UDMA xfer speed. if at UDMA0, switch to PIO4,
> - decrease PIO xfer speed. if at PIO3, complain, but continue

33.8 ata_piix Internals

int **ich_pata_cable_detect**(struct ata_port * ap)
Probe host controller cable detect info

Parameters

struct ata_port * ap Port for which cable detect info is desired

Read 80c cable indicator from ATA PCI device's PCI config register. This register is normally set by firmware (BIOS).

LOCKING: None (inherited from caller).

int **piix_pata_prereset**(struct ata_link * link, unsigned long deadline)
prereset for PATA host controller

Parameters

struct ata_link * link Target link

unsigned long deadline deadline jiffies for the operation

LOCKING: None (inherited from caller).

void **piix_set_piomode**(struct ata_port * ap, struct ata_device * adev)
Initialize host controller PATA PIO timings

Parameters

struct ata_port * ap Port whose timings we are configuring

struct ata_device * adev Drive in question

Set PIO mode for device, in host controller PCI config space.

LOCKING: None (inherited from caller).

void **do_pata_set_dmamode**(struct ata_port * ap, struct ata_device * adev, int isich)
Initialize host controller PATA PIO timings

Parameters

struct ata_port * ap Port whose timings we are configuring

struct ata_device * adev Drive in question

int isich set if the chip is an ICH device

Set UDMA mode for device, in host controller PCI config space.

LOCKING: None (inherited from caller).

void **piix_set_dmamode**(struct ata_port * ap, struct ata_device * adev)
Initialize host controller PATA DMA timings

Parameters

struct ata_port * ap Port whose timings we are configuring

struct ata_device * adev um

Set MW/UDMA mode for device, in host controller PCI config space.

LOCKING: None (inherited from caller).

void **ich_set_dmamode**(struct ata_port * ap, struct ata_device * adev)
Initialize host controller PATA DMA timings

Parameters

struct ata_port * ap Port whose timings we are configuring

struct ata_device * adev um

Set MW/UDMA mode for device, in host controller PCI config space.

LOCKING: None (inherited from caller).

int **piix_check_450nx_errata**(struct pci_dev * ata_dev)
Check for problem 450NX setup

Parameters

struct pci_dev * ata_dev the PCI device to check

Check for the present of 450NX errata #19 and errata #25. If they are found return an error code so we can turn off DMA

int **piix_init_one**(struct pci_dev * pdev, const struct pci_device_id * ent)
Register PIIX ATA PCI device with kernel services

Parameters

struct pci_dev * pdev PCI device to register

const struct pci_device_id * ent Entry in piix_pci_tbl matching with **pdev**

Called from kernel PCI layer. We probe for combined mode (sigh), and then hand over control to libata, for it to do the rest.

LOCKING: Inherited from PCI layer (may sleep).

Return

Zero on success, or -ERRNO value.

33.9 sata_sil Internals

int **sil_set_mode**(struct ata_link * link, struct ata_device ** r_failed)
wrap set_mode functions

Parameters

struct ata_link * link link to set up

struct ata_device ** r_failed returned device when we fail

Wrap the libata method for device setup as after the setup we need to inspect the results and do some configuration work

void **sil_dev_config**(struct ata_device * dev)
Apply device/host-specific errata fixups

Parameters

struct ata_device * dev Device to be examined

After the IDENTIFY [PACKET] DEVICE step is complete, and a device is known to be present, this function is called. We apply two errata fixups which are specific to Silicon Image, a Seagate and a Maxtor fixup.

For certain Seagate devices, we must limit the maximum sectors to under 8K.

For certain Maxtor devices, we must not program the drive beyond udma5.

Both fixups are unfairly pessimistic. As soon as I get more information on these errata, I will create a more exhaustive list, and apply the fixups to only the specific devices/hosts/firmwares that need it.

20040111 - Seagate drives affected by the Mod15Write bug are blacklisted The Maxtor quirk is in the blacklist, but I'm keeping the original pessimistic fix for the following reasons···- There seems to be less info on it, only one device gleaned off the Windows driver, maybe only one is affected. More info

would be greatly appreciated. - But then again UDMA5 is hardly anything to complain about

33.10 Thanks

The bulk of the ATA knowledge comes thanks to long conversations with Andre Hedrick (www.linux-ide.org), and long hours pondering the ATA and SCSI specifications.

Thanks to Alan Cox for pointing out similarities between SATA and SCSI, and in general for motivation to hack on libata.

libata's device detection method, ata_pio_devchk, and in general all the early probing was based on extensive study of Hale Landis's probe/reset code in his ATADRVR driver (www.ata-atapi.com).

CHAPTER

THIRTYFOUR

TARGET AND ISCSI INTERFACES GUIDE

34.1 Introduction and Overview

TBD

34.2 Target core device interfaces

This section is blank because no kerneldoc comments have been added to drivers/target/target_core_device.c.

34.3 Target core transport interfaces

int **transport_init_session**(struct se_session * se_sess)
initialize a session object

Parameters

struct se_session * se_sess Session object pointer.

Description

The caller must have zero-initialized **se_sess** before calling this function.

struct se_session * **transport_alloc_session**(enum target_prot_op sup_prot_ops)
allocate a session object and initialize it

Parameters

enum target_prot_op sup_prot_ops bitmask that defines which T10-PI modes are supported.

int **transport_alloc_session_tags**(struct se_session * se_sess, unsigned int tag_num, unsigned int tag_size)
allocate target driver private data

Parameters

struct se_session * se_sess Session pointer.

unsigned int tag_num Maximum number of in-flight commands between initiator and target.

unsigned int tag_size Size in bytes of the private data a target driver associates with each command.

int **target_submit_cmd_map_sgls**(struct se_cmd * se_cmd, struct se_session * se_sess, unsigned char * cdb, unsigned char * sense, u64 unpacked_lun, u32 data_length, int task_attr, int data_dir, int flags, struct scatterlist * sgl, u32 sgl_count, struct scatterlist * sgl_bidi, u32 sgl_bidi_count, struct scatterlist * sgl_prot, u32 sgl_prot_count)

lookup unpacked lun and submit uninitialized se_cmd + use pre-allocated SGL memory.

Parameters

struct se_cmd * se_cmd command descriptor to submit

struct se_session * se_sess associated se_sess for endpoint

unsigned char * cdb pointer to SCSI CDB

unsigned char * sense pointer to SCSI sense buffer

u64 unpacked_lun unpacked LUN to reference for struct se_lun

u32 data_length fabric expected data transfer length

int task_attr SAM task attribute

int data_dir DMA data direction

int flags flags for command submission from target_sc_flags_tables

struct scatterlist * sgl struct scatterlist memory for unidirectional mapping

u32 sgl_count scatterlist count for unidirectional mapping

struct scatterlist * sgl_bidi struct scatterlist memory for bidirectional READ mapping

u32 sgl_bidi_count scatterlist count for bidirectional READ mapping

struct scatterlist * sgl_prot struct scatterlist memory protection information

u32 sgl_prot_count scatterlist count for protection information

Description

Task tags are supported if the caller has set **se_cmd->tag**.

Returns non zero to signal active I/O shutdown failure. All other setup exceptions will be returned as a SCSI CHECK_CONDITION response, but still return zero here.

This may only be called from process context, and also currently assumes internal allocation of fabric payload buffer by target-core.

int **target_submit_cmd**(struct se_cmd * se_cmd, struct se_session * se_sess, unsigned char * cdb, unsigned char * sense, u64 unpacked_lun, u32 data_length, int task_attr, int data_dir, int flags)

lookup unpacked lun and submit uninitialized se_cmd

Parameters

struct se_cmd * se_cmd command descriptor to submit

struct se_session * se_sess associated se_sess for endpoint

unsigned char * cdb pointer to SCSI CDB

unsigned char * sense pointer to SCSI sense buffer

u64 unpacked_lun unpacked LUN to reference for struct se_lun

u32 data_length fabric expected data transfer length

int task_attr SAM task attribute

int data_dir DMA data direction

int flags flags for command submission from target_sc_flags_tables

Description

Task tags are supported if the caller has set **se_cmd->tag**.

Returns non zero to signal active I/O shutdown failure. All other setup exceptions will be returned as a SCSI CHECK_CONDITION response, but still return zero here.

This may only be called from process context, and also currently assumes internal allocation of fabric payload buffer by target-core.

It also assumes interal target core SGL memory allocation.

int **target_submit_tmr**(struct se_cmd * se_cmd, struct se_session * se_sess, unsigned char * sense, u64 unpacked_lun, void * fabric_tmr_ptr, unsigned char tm_type, gfp_t gfp, u64 tag, int flags)

lookup unpacked lun and submit uninitialized se_cmd for TMR CDBs

Parameters

struct se_cmd * se_cmd command descriptor to submit

struct se_session * se_sess associated se_sess for endpoint

unsigned char * sense pointer to SCSI sense buffer

u64 unpacked_lun unpacked LUN to reference for struct se_lun

void * fabric_tmr_ptr fabric context for TMR req

unsigned char tm_type Type of TM request

gfp_t gfp gfp type for caller

u64 tag referenced task tag for TMR_ABORT_TASK

int flags submit cmd flags

Description

Callable from all contexts.

int **target_get_sess_cmd**(struct se_cmd * se_cmd, bool ack_kref)
Add command to active ->sess_cmd_list

Parameters

struct se_cmd * se_cmd command descriptor to add

bool ack_kref Signal that fabric will perform an ack `target_put_sess_cmd()`

int **target_put_sess_cmd**(struct se_cmd * se_cmd)
decrease the command reference count

Parameters

struct se_cmd * se_cmd command to drop a reference from

Description

Returns 1 if and only if this `target_put_sess_cmd()` call caused the refcount to drop to zero. Returns zero otherwise.

void **target_sess_cmd_list_set_waiting**(struct se_session * se_sess)
Set sess_tearing_down so no new commands are queued.

Parameters

struct se_session * se_sess session to flag

void **target_wait_for_sess_cmds**(struct se_session * se_sess)
Wait for outstanding commands

Parameters

struct se_session * se_sess session to wait for active I/O

bool **transport_wait_for_tasks**(struct se_cmd * cmd)
set CMD_T_STOP and wait for t_transport_stop_comp

Parameters

struct se_cmd * cmd command to wait on

int **target_send_busy**(struct se_cmd * cmd)
Send SCSI BUSY status back to the initiator

Parameters

struct se_cmd * cmd SCSI command for which to send a BUSY reply.

Note

Only call this function if target_submit_cmd*() failed.

34.4 Target-supported userspace I/O

34.4.1 Userspace I/O

Define a shared-memory interface for LIO to pass SCSI commands and data to userspace for processing. This is to allow backends that are too complex for in-kernel support to be possible.

It uses the UIO framework to do a lot of the device-creation and introspection work for us.

See the .h file for how the ring is laid out. Note that while the command ring is defined, the particulars of the data area are not. Offset values in the command entry point to other locations internal to the mmap-ed area. There is separate space outside the command ring for data buffers. This leaves maximum flexibility for moving buffer allocations, or even page flipping or other allocation techniques, without altering the command ring layout.

SECURITY: The user process must be assumed to be malicious. There's no way to prevent it breaking the command ring protocol if it wants, but in order to prevent other issues we must only ever read data from the shared memory area, not offsets or sizes. This applies to command ring entries as well as the mailbox. Extra code needed for this may have a 'UAM' comment.

34.4.2 Ring Design

The mmaped area is divided into three parts: 1) The mailbox (struct tcmu_mailbox, below); 2) The command ring; 3) Everything beyond the command ring (data).

The mailbox tells userspace the offset of the command ring from the start of the shared memory region, and how big the command ring is.

The kernel passes SCSI commands to userspace by putting a struct tcmu_cmd_entry in the ring, updating mailbox->cmd_head, and poking userspace via UIO's interrupt mechanism.

tcmu_cmd_entry contains a header. If the header type is PAD, userspace should skip hdr->length bytes (mod cmdr_size) to find the next cmd_entry.

Otherwise, the entry will contain offsets into the mmaped area that contain the cdb and data buffers – the latter accessible via the iov array. iov addresses are also offsets into the shared area.

When userspace is completed handling the command, set entry->rsp.scsi_status, fill in rsp.sense_buffer if appropriate, and also set mailbox->cmd_tail equal to the old cmd_tail plus hdr->length, mod cmdr_size. If cmd_tail doesn't equal cmd_head, it should process the next packet the same way, and so on.

34.5 iSCSI helper functions

void **iscsi_prep_data_out_pdu**(struct iscsi_task * task, struct iscsi_r2t_info * r2t, struct iscsi_data * hdr)
initialize Data-Out

Parameters

struct iscsi_task * task scsi command task

struct iscsi_r2t_info * r2t R2T info

struct iscsi_data * hdr iscsi data in pdu

Notes

Initialize Data-Out within this R2T sequence and finds proper data_offset within this SCSI command.

This function is called with connection lock taken.

void **iscsi_complete_scsi_task**(struct iscsi_task * task, uint32_t exp_cmdsn, uint32_t max_cmdsn)
finish scsi task normally

Parameters

struct iscsi_task * task iscsi task for scsi cmd

uint32_t exp_cmdsn expected cmd sn in cpu format

uint32_t max_cmdsn max cmd sn in cpu format

Description

This is used when drivers do not need or cannot perform lower level pdu processing.

Called with session back_lock

struct iscsi_task * **iscsi_itt_to_task**(struct iscsi_conn * conn, itt_t itt)
look up task by itt

Parameters

struct iscsi_conn * conn iscsi connection

itt_t itt itt

Description

This should be used for mgmt tasks like login and nops, or if the LDD's itt space does not include the session age.

The session back_lock must be held.

int **__iscsi_complete_pdu**(struct iscsi_conn * conn, struct iscsi_hdr * hdr, char * data, int datalen)
complete pdu

Parameters

struct iscsi_conn * conn iscsi conn

struct iscsi_hdr * hdr iscsi header

char * data data buffer

int datalen len of data buffer

Description

Completes pdu processing by freeing any resources allocated at queuecommand or send generic. session back_lock must be held and verify itt must have been called.

struct iscsi_task * **iscsi_itt_to_ctask**(struct iscsi_conn * conn, itt_t itt)
look up ctask by itt

Parameters

struct iscsi_conn * conn iscsi connection

itt_t itt itt

Description

This should be used for cmd tasks.

The session back_lock must be held.

void **iscsi_requeue_task**(struct iscsi_task * task)
requeue task to run from session workqueue

Parameters

struct iscsi_task * task task to requeue

Description

LLDs that need to run a task from the session workqueue should call this. The session frwd_lock must be held. This should only be called by software drivers.

void **iscsi_suspend_queue**(struct iscsi_conn * conn)
suspend iscsi_queuecommand

Parameters

struct iscsi_conn * conn iscsi conn to stop queueing IO on

Description

This grabs the session frwd_lock to make sure no one is in xmit_task/queuecommand, and then sets suspend to prevent new commands from being queued. This only needs to be called by offload drivers that need to sync a path like ep disconnect with the iscsi_queuecommand/xmit_task. To start IO again libiscsi will call iscsi_start_tx and iscsi_unblock_session when in FFP.

void **iscsi_suspend_tx**(struct iscsi_conn * conn)
suspend iscsi_data_xmit

Parameters

struct iscsi_conn * conn iscsi conn tp stop processing IO on.

Description

This function sets the suspend bit to prevent iscsi_data_xmit from sending new IO, and if work is queued on the xmit thread it will wait for it to be completed.

int **iscsi_eh_session_reset**(struct scsi_cmnd * sc)
drop session and attempt relogin

Parameters

struct scsi_cmnd * sc scsi command

Description

This function will wait for a relogin, session termination from userspace, or a recovery/replacement timeout.

int **iscsi_eh_recover_target**(struct scsi_cmnd * sc)
reset target and possibly the session

Parameters

struct scsi_cmnd * sc scsi command

Description

This will attempt to send a warm target reset. If that fails, we will escalate to ERL0 session recovery.

int **iscsi_host_add**(struct Scsi_Host * shost, struct device * pdev)
add host to system

Parameters

struct Scsi_Host * shost scsi host

struct device * pdev parent device

Description

This should be called by partial offload and software iscsi drivers to add a host to the system.

struct Scsi_Host * **iscsi_host_alloc**(struct scsi_host_template * sht, int dd_data_size, bool xmit_can_sleep)
allocate a host and driver data

Parameters

struct scsi_host_template * sht scsi host template

int dd_data_size driver host data size

bool xmit_can_sleep bool indicating if LLD will queue IO from a work queue

Description

This should be called by partial offload and software iscsi drivers. To access the driver specific memory use the iscsi_host_priv() macro.

void **iscsi_host_remove**(struct Scsi_Host * shost)
remove host and sessions

Parameters

struct Scsi_Host * shost scsi host

Description

If there are any sessions left, this will initiate the removal and wait for the completion.

struct iscsi_cls_session * **iscsi_session_setup**(struct iscsi_transport * iscsit, struct Scsi_Host * shost, uint16_t cmds_max, int dd_size, int cmd_task_size, uint32_t initial_cmdsn, unsigned int id)
create iscsi cls session and host and session

Parameters

struct iscsi_transport * iscsit iscsi transport template

struct Scsi_Host * shost scsi host

uint16_t cmds_max session can queue

int dd_size private driver data size, added to session allocation size

int cmd_task_size LLD task private data size

uint32_t initial_cmdsn initial CmdSN

unsigned int id target ID to add to this session

Description

This can be used by software iscsi_transports that allocate a session per scsi host.

Callers should set cmds_max to the largest total numer (mgmt + scsi) of tasks they support. The iscsi layer reserves ISCSI_MGMT_CMDS_MAX tasks for nop handling and login/logout requests.

void **iscsi_session_teardown**(struct iscsi_cls_session * cls_session)
destroy session, host, and cls_session

Parameters

struct iscsi_cls_session * cls_session iscsi session

struct iscsi_cls_conn * **iscsi_conn_setup**(struct iscsi_cls_session * cls_session, int dd_size, uint32_t conn_idx)
create iscsi_cls_conn and iscsi_conn

Parameters

struct iscsi_cls_session * cls_session iscsi_cls_session

int dd_size private driver data size

uint32_t conn_idx cid

void **iscsi_conn_teardown**(struct iscsi_cls_conn * cls_conn)
teardown iscsi connection

Parameters

struct iscsi_cls_conn * cls_conn iscsi class connection

Description

TODO: we may need to make this into a two step process like scsi-mls remove + put host

34.6 iSCSI boot information

struct iscsi_boot_kobj * **iscsi_boot_create_target**(struct iscsi_boot_kset * boot_kset, int index, void * data, ssize_t (*show)(void *data, int type, char *buf), umode_t (*is_visible) (void *data, int type), void (*release) (void *data))

create boot target sysfs dir

Parameters

struct iscsi_boot_kset * boot_kset boot kset

int index the target id

void * data driver specific data for target

ssize_t (*) (void *data, int type, char *buf) show attr show function

umode_t (*) (void *data, int type) is_visible attr visibility function

void (*) (void *data) release release function

Note

The boot sysfs lib will free the data passed in for the caller when all refs to the target kobject have been released.

struct iscsi_boot_kobj * **iscsi_boot_create_initiator**(struct iscsi_boot_kset * boot_kset, int index, void * data, ssize_t (*show)(void *data, int type, char *buf), umode_t (*is_visible) (void *data, int type), void (*release) (void *data))

create boot initiator sysfs dir

Parameters

struct iscsi_boot_kset * boot_kset boot kset

int index the initiator id

void * data driver specific data

ssize_t (*) (void *data, int type, char *buf) show attr show function

umode_t (*) (void *data, int type) is_visible attr visibility function

void (*) (void *data) release release function

Note

The boot sysfs lib will free the data passed in for the caller when all refs to the initiator kobject have been released.

struct iscsi_boot_kobj * **iscsi_boot_create_ethernet**(struct iscsi_boot_kset * boot_kset, int index, void * data, ssize_t (*show)(void *data, int type, char *buf), umode_t (*is_visible) (void *data, int type), void (*release) (void *data))

create boot ethernet sysfs dir

Parameters

struct iscsi_boot_kset * boot_kset boot kset

int index the ethernet device id

void * data driver specific data

ssize_t (*) (void *data, int type, char *buf) show attr show function

umode_t (*) (void *data, int type) is_visible attr visibility function

void (*) (void *data) release release function

Note

The boot sysfs lib will free the data passed in for the caller when all refs to the ethernet kobject have been released.

struct iscsi_boot_kobj * **iscsi_boot_create_acpitbl**(struct iscsi_boot_kset * boot_kset, int index, void * data, ssize_t (*show)(void *data, int type, char *buf), umode_t (*is_visible) (void *data, int type), void (*release) (void *data))

create boot acpi table sysfs dir

Parameters

struct iscsi_boot_kset * boot_kset boot kset

int index not used

void * data driver specific data

ssize_t (*)(void *data, int type, char *buf) show attr show function

umode_t (*)(void *data, int type) is_visible attr visibility function

void (*)(void *data) release release function

Note

The boot sysfs lib will free the data passed in for the caller when all refs to the acpitbl kobject have been released.

struct iscsi_boot_kset * **iscsi_boot_create_kset**(const char * set_name)
creates root sysfs tree

Parameters

const char * set_name name of root dir

struct iscsi_boot_kset * **iscsi_boot_create_host_kset**(unsigned int hostno)
creates root sysfs tree for a scsi host

Parameters

unsigned int hostno host number of scsi host

void **iscsi_boot_destroy_kset**(struct iscsi_boot_kset * boot_kset)
destroy kset and kobjects under it

Parameters

struct iscsi_boot_kset * boot_kset boot kset

Description

This will remove the kset and kobjects and attrs under it.

34.7 iSCSI transport class

The file drivers/scsi/scsi_transport_iscsi.c defines transport attributes for the iSCSI class, which sends SCSI packets over TCP/IP connections.

struct iscsi_bus_flash_session * **iscsi_create_flashnode_sess**(struct Scsi_Host * shost, int index, struct iscsi_transport * transport, int dd_size)
Add flashnode session entry in sysfs

Parameters

struct Scsi_Host * shost pointer to host data

int index index of flashnode to add in sysfs

struct iscsi_transport * transport pointer to transport data

int dd_size total size to allocate

Description

Adds a sysfs entry for the flashnode session attributes

Return

pointer to allocated flashnode sess on success NULL on failure

struct iscsi_bus_flash_conn * **iscsi_create_flashnode_conn**(struct Scsi_Host * shost, struct iscsi_bus_flash_session * fnode_sess, struct iscsi_transport * transport, int dd_size)

Add flashnode conn entry in sysfs

Parameters

struct Scsi_Host * shost pointer to host data

struct iscsi_bus_flash_session * fnode_sess pointer to the parent flashnode session entry

struct iscsi_transport * transport pointer to transport data

int dd_size total size to allocate

Description

Adds a sysfs entry for the flashnode connection attributes

Return

pointer to allocated flashnode conn on success NULL on failure

struct device * **iscsi_find_flashnode_sess**(struct Scsi_Host * shost, void * data, int (*fn)(struct device *dev, void *data))

finds flashnode session entry

Parameters

struct Scsi_Host * shost pointer to host data

void * data pointer to data containing value to use for comparison

int (*)(struct device *dev, void *data) fn function pointer that does actual comparison

Description

Finds the flashnode session object comparing the data passed using logic defined in passed function pointer

Return

pointer to found flashnode session device object on success NULL on failure

struct device * **iscsi_find_flashnode_conn**(struct iscsi_bus_flash_session * fnode_sess)
finds flashnode connection entry

Parameters

struct iscsi_bus_flash_session * fnode_sess pointer to parent flashnode session entry

Description

Finds the flashnode connection object comparing the data passed using logic defined in passed function pointer

Return

pointer to found flashnode connection device object on success NULL on failure

void **iscsi_destroy_flashnode_sess**(struct iscsi_bus_flash_session * fnode_sess)
destroy flashnode session entry

Parameters

struct iscsi_bus_flash_session * fnode_sess pointer to flashnode session entry to be destroyed

Description

Deletes the flashnode session entry and all children flashnode connection entries from sysfs

void **iscsi_destroy_all_flashnode**(struct Scsi_Host * shost)
destroy all flashnode session entries

Parameters

struct Scsi_Host * shost pointer to host data

Description

Destroys all the flashnode session entries and all corresponding children flashnode connection entries from sysfs

int **iscsi_scan_finished**(struct Scsi_Host * shost, unsigned long time)
helper to report when running scans are done

Parameters

struct Scsi_Host * shost scsi host

unsigned long time scan run time

Description

This function can be used by drives like qla4xxx to report to the scsi layer when the scans it kicked off at module load time are done.

int **iscsi_block_scsi_eh**(struct scsi_cmnd * cmd)
block scsi eh until session state has transistioned

Parameters

struct scsi_cmnd * cmd scsi cmd passed to scsi eh handler

Description

If the session is down this function will wait for the recovery timer to fire or for the session to be logged back in. If the recovery timer fires then FAST_IO_FAIL is returned. The caller should pass this error value to the scsi eh.

void **iscsi_unblock_session**(struct iscsi_cls_session * session)
set a session as logged in and start IO.

Parameters

struct iscsi_cls_session * session iscsi session

Description

Mark a session as ready to accept IO.

struct iscsi_cls_session * **iscsi_create_session**(struct Scsi_Host * shost, struct iscsi_transport * transport, int dd_size, unsigned int target_id)
create iscsi class session

Parameters

struct Scsi_Host * shost scsi host

struct iscsi_transport * transport iscsi transport

int dd_size private driver data size

unsigned int target_id which target

Description

This can be called from a LLD or iscsi_transport.

struct iscsi_cls_conn * **iscsi_create_conn**(struct iscsi_cls_session * session, int dd_size, uint32_t cid)
create iscsi class connection

Parameters

struct iscsi_cls_session * session iscsi cls session

int dd_size private driver data size

uint32_t cid connection id

Description

This can be called from a LLD or iscsi_transport. The connection is child of the session so cid must be unique for all connections on the session.

Since we do not support MCS, cid will normally be zero. In some cases for software iscsi we could be trying to preallocate a connection struct in which case there could be two connection structs and cid would be non-zero.

int **iscsi_destroy_conn**(struct iscsi_cls_conn * conn)
destroy iscsi class connection

Parameters

struct iscsi_cls_conn * conn iscsi cls session

Description

This can be called from a LLD or iscsi_transport.

int **iscsi_session_event**(struct iscsi_cls_session * session, enum iscsi_uevent_e event)
send session destr. completion event

Parameters

struct iscsi_cls_session * session iscsi class session

enum iscsi_uevent_e event type of event

34.8 iSCSI TCP interfaces

int **iscsi_sw_tcp_recv**(read_descriptor_t * rd_desc, struct sk_buff * skb, unsigned int offset, size_t len)
TCP receive in sendfile fashion

Parameters

read_descriptor_t * rd_desc read descriptor

struct sk_buff * skb socket buffer

unsigned int offset offset in skb

size_t len skb->len - offset

int **iscsi_sw_sk_state_check**(struct sock * sk)
check socket state

Parameters

struct sock * sk socket

Description

If the socket is in CLOSE or CLOSE_WAIT we should not close the connection if there is still some data pending.

Must be called with sk_callback_lock.

void **iscsi_sw_tcp_write_space**(struct sock * sk)
Called when more output buffer space is available

Parameters

struct sock * sk socket space is available for

int **iscsi_sw_tcp_xmit_segment**(struct iscsi_tcp_conn * tcp_conn, struct iscsi_segment * segment)
transmit segment

Parameters

struct iscsi_tcp_conn * tcp_conn the iSCSI TCP connection

struct iscsi_segment * segment the buffer to transmnit

Description

This function transmits as much of the buffer as the network layer will accept, and returns the number of bytes transmitted.

If CRC hashing is enabled, the function will compute the hash as it goes. When the entire segment has been transmitted, it will retrieve the hash value and send it as well.

int **iscsi_sw_tcp_xmit**(struct iscsi_conn * conn)
TCP transmit

Parameters

struct iscsi_conn * conn iscsi connection

int **iscsi_sw_tcp_xmit_qlen**(struct iscsi_conn * conn)
return the number of bytes queued for xmit

Parameters

struct iscsi_conn * conn iscsi connection

int **iscsi_tcp_segment_done**(struct iscsi_tcp_conn * tcp_conn, struct iscsi_segment * segment, int recv, unsigned copied)
check whether the segment is complete

Parameters

struct iscsi_tcp_conn * tcp_conn iscsi tcp connection

struct iscsi_segment * segment iscsi segment to check

int recv set to one of this is called from the recv path

unsigned copied number of bytes copied

Description

Check if we're done receiving this segment. If the receive buffer is full but we expect more data, move on to the next entry in the scatterlist.

If the amount of data we received isn't a multiple of 4, we will transparently receive the pad bytes, too.

This function must be re-entrant.

void **iscsi_tcp_hdr_recv_prep**(struct iscsi_tcp_conn * tcp_conn)
prep segment for hdr reception

Parameters

struct iscsi_tcp_conn * tcp_conn iscsi connection to prep for

Description

This function always passes NULL for the hash argument, because when this function is called we do not yet know the final size of the header and want to delay the digest processing until we know that.

void **iscsi_tcp_cleanup_task**(struct iscsi_task * task)
free tcp_task resources

Parameters

struct iscsi_task * task iscsi task

Description

must be called with session back_lock

int **iscsi_tcp_recv_segment_is_hdr**(struct iscsi_tcp_conn * tcp_conn)
tests if we are reading in a header

Parameters

struct iscsi_tcp_conn * tcp_conn iscsi tcp conn

Description

returns non zero if we are currently processing or setup to process a header.

int **iscsi_tcp_recv_skb**(struct iscsi_conn * conn, struct sk_buff * skb, unsigned int offset, bool offloaded, int * status)
Process skb

Parameters

struct iscsi_conn * conn iscsi connection

struct sk_buff * skb network buffer with header and/or data segment

unsigned int offset offset in skb

bool offloaded bool indicating if transfer was offloaded

int * status iscsi TCP status result

Description

Will return status of transfer in **status**. And will return number of bytes copied.

int **iscsi_tcp_task_init**(struct iscsi_task * task)
Initialize iSCSI SCSI_READ or SCSI_WRITE commands

Parameters

struct iscsi_task * task scsi command task

int **iscsi_tcp_task_xmit**(struct iscsi_task * task)
xmit normal PDU task

Parameters

struct iscsi_task * task iscsi command task

Description

We’re expected to return 0 when everything was transmitted successfully, -EAGAIN if there’s still data in the queue, or != 0 for any other kind of error.

CHAPTER

THIRTYFIVE

MTD NAND DRIVER PROGRAMMING INTERFACE

Author Thomas Gleixner

35.1 Introduction

The generic NAND driver supports almost all NAND and AG-AND based chips and connects them to the Memory Technology Devices (MTD) subsystem of the Linux Kernel.

This documentation is provided for developers who want to implement board drivers or filesystem drivers suitable for NAND devices.

35.2 Known Bugs And Assumptions

None.

35.3 Documentation hints

The function and structure docs are autogenerated. Each function and struct member has a short description which is marked with an [XXX] identifier. The following chapters explain the meaning of those identifiers.

35.3.1 Function identifiers [XXX]

The functions are marked with [XXX] identifiers in the short comment. The identifiers explain the usage and scope of the functions. Following identifiers are used:

- [MTD Interface]

  These functions provide the interface to the MTD kernel API. They are not replaceable and provide functionality which is complete hardware independent.

- [NAND Interface]

  These functions are exported and provide the interface to the NAND kernel API.

- [GENERIC]

  Generic functions are not replaceable and provide functionality which is complete hardware independent.

- [DEFAULT]

  Default functions provide hardware related functionality which is suitable for most of the implementations. These functions can be replaced by the board driver if necessary. Those functions are called via pointers in the NAND chip description structure. The board driver can set the functions which should be replaced by board dependent functions before calling nand_scan(). If the function pointer is NULL on entry to nand_scan() then the pointer is set to the default function which is suitable for the detected chip type.

35.3.2 Struct member identifiers [XXX]

The struct members are marked with [XXX] identifiers in the comment. The identifiers explain the usage and scope of the members. Following identifiers are used:

- [INTERN]

  These members are for NAND driver internal use only and must not be modified. Most of these values are calculated from the chip geometry information which is evaluated during nand_scan().

- [REPLACEABLE]

  Replaceable members hold hardware related functions which can be provided by the board driver. The board driver can set the functions which should be replaced by board dependent functions before calling nand_scan(). If the function pointer is NULL on entry to nand_scan() then the pointer is set to the default function which is suitable for the detected chip type.

- [BOARDSPECIFIC]

  Board specific members hold hardware related information which must be provided by the board driver. The board driver must set the function pointers and datafields before calling nand_scan().

- [OPTIONAL]

  Optional members can hold information relevant for the board driver. The generic NAND driver code does not use this information.

35.4 Basic board driver

For most boards it will be sufficient to provide just the basic functions and fill out some really board dependent members in the nand chip description structure.

35.4.1 Basic defines

At least you have to provide a nand_chip structure and a storage for the ioremap'ed chip address. You can allocate the nand_chip structure using kmalloc or you can allocate it statically. The NAND chip structure embeds an mtd structure which will be registered to the MTD subsystem. You can extract a pointer to the mtd structure from a nand_chip pointer using the nand_to_mtd() helper.

Kmalloc based example

```
static struct mtd_info *board_mtd;
static void __iomem *baseaddr;
```

Static example

```
static struct nand_chip board_chip;
static void __iomem *baseaddr;
```

35.4.2 Partition defines

If you want to divide your device into partitions, then define a partitioning scheme suitable to your board.

```
#define NUM_PARTITIONS 2
static struct mtd_partition partition_info[] = {
    { .name = "Flash partition 1",
      .offset =  0,
      .size =    8 * 1024 * 1024 },
    { .name = "Flash partition 2",
      .offset =  MTDPART_OFS_NEXT,
      .size =    MTDPART_SIZ_FULL },
};
```

35.4.3 Hardware control function

The hardware control function provides access to the control pins of the NAND chip(s). The access can be done by GPIO pins or by address lines. If you use address lines, make sure that the timing requirements are met.

GPIO based example

```
static void board_hwcontrol(struct mtd_info *mtd, int cmd)
{
    switch(cmd){
        case NAND_CTL_SETCLE: /* Set CLE pin high */ break;
        case NAND_CTL_CLRCLE: /* Set CLE pin low */ break;
        case NAND_CTL_SETALE: /* Set ALE pin high */ break;
        case NAND_CTL_CLRALE: /* Set ALE pin low */ break;
        case NAND_CTL_SETNCE: /* Set nCE pin low */ break;
        case NAND_CTL_CLRNCE: /* Set nCE pin high */ break;
    }
}
```

Address lines based example. It' s assumed that the nCE pin is driven by a chip select decoder.

```
static void board_hwcontrol(struct mtd_info *mtd, int cmd)
{
    struct nand_chip *this = mtd_to_nand(mtd);
    switch(cmd){
        case NAND_CTL_SETCLE: this->legacy.IO_ADDR_W |= CLE_ADRR_BIT; ␣
↪break;
        case NAND_CTL_CLRCLE: this->legacy.IO_ADDR_W &= ~CLE_ADRR_BIT;␣
↪break;
        case NAND_CTL_SETALE: this->legacy.IO_ADDR_W |= ALE_ADRR_BIT; ␣
↪break;
        case NAND_CTL_CLRALE: this->legacy.IO_ADDR_W &= ~ALE_ADRR_BIT;␣
↪break;
    }
}
```

35.4.4 Device ready function

If the hardware interface has the ready busy pin of the NAND chip connected to a GPIO or other accessible I/O pin, this function is used to read back the state of the pin. The function has no arguments and should return 0, if the device is busy (R/B pin is low) and 1, if the device is ready (R/B pin is high). If the hardware interface does not give access to the ready busy pin, then the function must not be defined and the function pointer this->legacy.dev_ready is set to NULL.

35.4.5 Init function

The init function allocates memory and sets up all the board specific parameters and function pointers. When everything is set up nand_scan() is called. This function tries to detect and identify then chip. If a chip is found all the internal data fields are initialized accordingly. The structure(s) have to be zeroed out first and then filled with the necessary information about the device.

```
static int __init board_init (void)
{
    struct nand_chip *this;
    int err = 0;

    /* Allocate memory for MTD device structure and private data */
    this = kzalloc(sizeof(struct nand_chip), GFP_KERNEL);
    if (!this) {
        printk ("Unable to allocate NAND MTD device structure.\n");
        err = -ENOMEM;
        goto out;
    }

    board_mtd = nand_to_mtd(this);

    /* map physical address */
    baseaddr = ioremap(CHIP_PHYSICAL_ADDRESS, 1024);
    if (!baseaddr) {
```

(continues on next page)

(continued from previous page)

```
        printk("Ioremap to access NAND chip failed\n");
        err = -EIO;
        goto out_mtd;
    }

    /* Set address of NAND IO lines */
    this->legacy.IO_ADDR_R = baseaddr;
    this->legacy.IO_ADDR_W = baseaddr;
    /* Reference hardware control function */
    this->hwcontrol = board_hwcontrol;
    /* Set command delay time, see datasheet for correct value */
    this->legacy.chip_delay = CHIP_DEPENDEND_COMMAND_DELAY;
    /* Assign the device ready function, if available */
    this->legacy.dev_ready = board_dev_ready;
    this->eccmode = NAND_ECC_SOFT;

    /* Scan to find existence of the device */
    if (nand_scan (this, 1)) {
        err = -ENXIO;
        goto out_ior;
    }

    add_mtd_partitions(board_mtd, partition_info, NUM_PARTITIONS);
    goto out;

out_ior:
    iounmap(baseaddr);
out_mtd:
    kfree (this);
out:
    return err;
}
module_init(board_init);
```

35.4.6 Exit function

The exit function is only necessary if the driver is compiled as a module. It releases all resources which are held by the chip driver and unregisters the partitions in the MTD layer.

```
#ifdef MODULE
static void __exit board_cleanup (void)
{
    /* Unregister device */
    WARN_ON(mtd_device_unregister(board_mtd));
    /* Release resources */
    nand_cleanup(mtd_to_nand(board_mtd));

    /* unmap physical address */
    iounmap(baseaddr);

    /* Free the MTD device structure */
    kfree (mtd_to_nand(board_mtd));
}
```

(continues on next page)

(continued from previous page)

```
module_exit(board_cleanup);
#endif
```

35.5 Advanced board driver functions

This chapter describes the advanced functionality of the NAND driver. For a list of functions which can be overridden by the board driver see the documentation of the nand_chip structure.

35.5.1 Multiple chip control

The nand driver can control chip arrays. Therefore the board driver must provide an own select_chip function. This function must (de)select the requested chip. The function pointer in the nand_chip structure must be set before calling nand_scan(). The maxchip parameter of nand_scan() defines the maximum number of chips to scan for. Make sure that the select_chip function can handle the requested number of chips.

The nand driver concatenates the chips to one virtual chip and provides this virtual chip to the MTD layer.

Note: The driver can only handle linear chip arrays of equally sized chips. There is no support for parallel arrays which extend the buswidth.

GPIO based example

```
static void board_select_chip (struct mtd_info *mtd, int chip)
{
    /* Deselect all chips, set all nCE pins high */
    GPIO(BOARD_NAND_NCE) |= 0xff;
    if (chip >= 0)
        GPIO(BOARD_NAND_NCE) &= ~ (1 << chip);
}
```

Address lines based example. Its assumed that the nCE pins are connected to an address decoder.

```
static void board_select_chip (struct mtd_info *mtd, int chip)
{
    struct nand_chip *this = mtd_to_nand(mtd);

    /* Deselect all chips */
    this->legacy.IO_ADDR_R &= ~BOARD_NAND_ADDR_MASK;
    this->legacy.IO_ADDR_W &= ~BOARD_NAND_ADDR_MASK;
    switch (chip) {
    case 0:
        this->legacy.IO_ADDR_R |= BOARD_NAND_ADDR_CHIP0;
        this->legacy.IO_ADDR_W |= BOARD_NAND_ADDR_CHIP0;
        break;
    ....
    case n:
```

(continues on next page)

(continued from previous page)

```
        this->legacy.IO_ADDR_R |= BOARD_NAND_ADDR_CHIPn;
        this->legacy.IO_ADDR_W |= BOARD_NAND_ADDR_CHIPn;
        break;
    }
}
```

35.5.2 Hardware ECC support

Functions and constants

The nand driver supports three different types of hardware ECC.

- NAND_ECC_HW3_256

  Hardware ECC generator providing 3 bytes ECC per 256 byte.

- NAND_ECC_HW3_512

  Hardware ECC generator providing 3 bytes ECC per 512 byte.

- NAND_ECC_HW6_512

  Hardware ECC generator providing 6 bytes ECC per 512 byte.

- NAND_ECC_HW8_512

  Hardware ECC generator providing 8 bytes ECC per 512 byte.

If your hardware generator has a different functionality add it at the appropriate place in nand_base.c

The board driver must provide following functions:

- enable_hwecc

  This function is called before reading / writing to the chip. Reset or initialize the hardware generator in this function. The function is called with an argument which let you distinguish between read and write operations.

- calculate_ecc

  This function is called after read / write from / to the chip. Transfer the ECC from the hardware to the buffer. If the option NAND_HWECC_SYNDROME is set then the function is only called on write. See below.

- correct_data

  In case of an ECC error this function is called for error detection and correction. Return 1 respectively 2 in case the error can be corrected. If the error is not correctable return -1. If your hardware generator matches the default algorithm of the nand_ecc software generator then use the correction function provided by nand_ecc instead of implementing duplicated code.

Hardware ECC with syndrome calculation

Many hardware ECC implementations provide Reed-Solomon codes and calculate an error syndrome on read. The syndrome must be converted to a standard Reed-Solomon syndrome before calling the error correction code in the generic Reed-Solomon library.

The ECC bytes must be placed immediately after the data bytes in order to make the syndrome generator work. This is contrary to the usual layout used by software ECC. The separation of data and out of band area is not longer possible. The nand driver code handles this layout and the remaining free bytes in the oob area are managed by the autoplacement code. Provide a matching oob-layout in this case. See rts_from4.c and diskonchip.c for implementation reference. In those cases we must also use bad block tables on FLASH, because the ECC layout is interfering with the bad block marker positions. See bad block table support for details.

35.5.3 Bad block table support

Most NAND chips mark the bad blocks at a defined position in the spare area. Those blocks must not be erased under any circumstances as the bad block information would be lost. It is possible to check the bad block mark each time when the blocks are accessed by reading the spare area of the first page in the block. This is time consuming so a bad block table is used.

The nand driver supports various types of bad block tables.

- Per device

  The bad block table contains all bad block information of the device which can consist of multiple chips.

- Per chip

  A bad block table is used per chip and contains the bad block information for this particular chip.

- Fixed offset

  The bad block table is located at a fixed offset in the chip (device). This applies to various DiskOnChip devices.

- Automatic placed

  The bad block table is automatically placed and detected either at the end or at the beginning of a chip (device)

- Mirrored tables

  The bad block table is mirrored on the chip (device) to allow updates of the bad block table without data loss.

nand_scan() calls the function nand_default_bbt(). nand_default_bbt() selects appropriate default bad block table descriptors depending on the chip information which was retrieved by nand_scan().

The standard policy is scanning the device for bad blocks and build a ram based bad block table which allows faster access than always checking the bad block information on the flash chip itself.

Flash based tables

It may be desired or necessary to keep a bad block table in FLASH. For AG-AND chips this is mandatory, as they have no factory marked bad blocks. They have factory marked good blocks. The marker pattern is erased when the block is erased to be reused. So in case of powerloss before writing the pattern back to the chip this block would be lost and added to the bad blocks. Therefore we scan the chip(s) when we detect them the first time for good blocks and store this information in a bad block table before erasing any of the blocks.

The blocks in which the tables are stored are protected against accidental access by marking them bad in the memory bad block table. The bad block table management functions are allowed to circumvent this protection.

The simplest way to activate the FLASH based bad block table support is to set the option NAND_BBT_USE_FLASH in the bbt_option field of the nand chip structure before calling nand_scan(). For AG-AND chips is this done by default. This activates the default FLASH based bad block table functionality of the NAND driver. The default bad block table options are

- Store bad block table per chip
- Use 2 bits per block
- Automatic placement at the end of the chip
- Use mirrored tables with version numbers
- Reserve 4 blocks at the end of the chip

User defined tables

User defined tables are created by filling out a nand_bbt_descr structure and storing the pointer in the nand_chip structure member bbt_td before calling nand_scan(). If a mirror table is necessary a second structure must be created and a pointer to this structure must be stored in bbt_md inside the nand_chip structure. If the bbt_md member is set to NULL then only the main table is used and no scan for the mirrored table is performed.

The most important field in the nand_bbt_descr structure is the options field. The options define most of the table properties. Use the predefined constants from rawnand.h to define the options.

- Number of bits per block

  The supported number of bits is 1, 2, 4, 8.

- Table per chip

  Setting the constant NAND_BBT_PERCHIP selects that a bad block table is managed for each chip in a chip array. If this option is not set then a per device bad block table is used.

- Table location is absolute

  Use the option constant NAND_BBT_ABSPAGE and define the absolute page number where the bad block table starts in the field pages. If you have selected bad block tables per chip and you have a multi chip array then the start

page must be given for each chip in the chip array. Note: there is no scan for a table ident pattern performed, so the fields pattern, veroffs, offs, len can be left uninitialized

- Table location is automatically detected

  The table can either be located in the first or the last good blocks of the chip (device). Set NAND_BBT_LASTBLOCK to place the bad block table at the end of the chip (device). The bad block tables are marked and identified by a pattern which is stored in the spare area of the first page in the block which holds the bad block table. Store a pointer to the pattern in the pattern field. Further the length of the pattern has to be stored in len and the offset in the spare area must be given in the offs member of the nand_bbt_descr structure. For mirrored bad block tables different patterns are mandatory.

- Table creation

  Set the option NAND_BBT_CREATE to enable the table creation if no table can be found during the scan. Usually this is done only once if a new chip is found.

- Table write support

  Set the option NAND_BBT_WRITE to enable the table write support. This allows the update of the bad block table(s) in case a block has to be marked bad due to wear. The MTD interface function block_markbad is calling the update function of the bad block table. If the write support is enabled then the table is updated on FLASH.

  Note: Write support should only be enabled for mirrored tables with version control.

- Table version control

  Set the option NAND_BBT_VERSION to enable the table version control. It’ s highly recommended to enable this for mirrored tables with write support. It makes sure that the risk of losing the bad block table information is reduced to the loss of the information about the one worn out block which should be marked bad. The version is stored in 4 consecutive bytes in the spare area of the device. The position of the version number is defined by the member veroffs in the bad block table descriptor.

- Save block contents on write

  In case that the block which holds the bad block table does contain other useful information, set the option NAND_BBT_SAVECONTENT. When the bad block table is written then the whole block is read the bad block table is updated and the block is erased and everything is written back. If this option is not set only the bad block table is written and everything else in the block is ignored and erased.

- Number of reserved blocks

  For automatic placement some blocks must be reserved for bad block table storage. The number of reserved blocks is defined in the maxblocks member of the bad block table description structure. Reserving 4 blocks for mirrored tables should be a reasonable number. This also limits the number of blocks which are scanned for the bad block table ident pattern.

35.5.4 Spare area (auto)placement

The nand driver implements different possibilities for placement of filesystem data in the spare area,

- Placement defined by fs driver
- Automatic placement

The default placement function is automatic placement. The nand driver has built in default placement schemes for the various chiptypes. If due to hardware ECC functionality the default placement does not fit then the board driver can provide a own placement scheme.

File system drivers can provide a own placement scheme which is used instead of the default placement scheme.

Placement schemes are defined by a nand_oobinfo structure

```
struct nand_oobinfo {
    int useecc;
    int eccbytes;
    int eccpos[24];
    int oobfree[8][2];
};
```

- useecc

  The useecc member controls the ecc and placement function. The header file include/mtd/mtd-abi.h contains constants to select ecc and placement. MTD_NANDECC_OFF switches off the ecc complete. This is not recommended and available for testing and diagnosis only. MTD_NANDECC_PLACE selects caller defined placement, MTD_NANDECC_AUTOPLACE selects automatic placement.

- eccbytes

  The eccbytes member defines the number of ecc bytes per page.

- eccpos

  The eccpos array holds the byte offsets in the spare area where the ecc codes are placed.

- oobfree

  The oobfree array defines the areas in the spare area which can be used for automatic placement. The information is given in the format {offset, size}. offset defines the start of the usable area, size the length in bytes. More than one area can be defined. The list is terminated by an {0, 0} entry.

Placement defined by fs driver

The calling function provides a pointer to a nand_oobinfo structure which defines the ecc placement. For writes the caller must provide a spare area buffer along with the data buffer. The spare area buffer size is (number of pages) * (size of spare area). For reads the buffer size is (number of pages) * ((size of spare area) + (number of ecc steps per page) * sizeof (int)). The driver stores the result of the ecc check for each tuple in the spare buffer. The storage sequence is:

```
<spare data page 0><ecc result 0>...<ecc result n>

...

<spare data page n><ecc result 0>...<ecc result n>
```

This is a legacy mode used by YAFFS1.

If the spare area buffer is NULL then only the ECC placement is done according to the given scheme in the nand_oobinfo structure.

Automatic placement

Automatic placement uses the built in defaults to place the ecc bytes in the spare area. If filesystem data have to be stored / read into the spare area then the calling function must provide a buffer. The buffer size per page is determined by the oobfree array in the nand_oobinfo structure.

If the spare area buffer is NULL then only the ECC placement is done according to the default builtin scheme.

35.5.5 Spare area autoplacement default schemes

256 byte pagesize

Off-set	Content	Comment
0x00	ECC byte 0	Error correction code byte 0
0x01	ECC byte 1	Error correction code byte 1
0x02	ECC byte 2	Error correction code byte 2
0x03	Auto-place 0	
0x04	Auto-place 1	
0x05	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x06	Auto-place 2	
0x07	Auto-place 3	

512 byte pagesize

Off-set	Con-tent	Comment
0x00	ECC byte 0	Error correction code byte 0 of the lower 256 Byte data in this page
0x01	ECC byte 1	Error correction code byte 1 of the lower 256 Bytes of data in this page
0x02	ECC byte 2	Error correction code byte 2 of the lower 256 Bytes of data in this page
0x03	ECC byte 3	Error correction code byte 0 of the upper 256 Bytes of data in this page
0x04	re-served	reserved
0x05	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x06	ECC byte 4	Error correction code byte 1 of the upper 256 Bytes of data in this page
0x07	ECC byte 5	Error correction code byte 2 of the upper 256 Bytes of data in this page
0x08 - 0x0F	Auto-place 0 - 7	

2048 byte pagesize

Off-set	Con-tent	Comment
0x00	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x01	Re-served	Reserved
0x02-0x27	Auto-place 0 - 37	
0x28	ECC byte 0	Error correction code byte 0 of the first 256 Byte data in this page
0x29	ECC byte 1	Error correction code byte 1 of the first 256 Bytes of data in this page
0x2A	ECC byte 2	Error correction code byte 2 of the first 256 Bytes data in this page
0x2B	ECC byte 3	Error correction code byte 0 of the second 256 Bytes of data in this page
0x2C	ECC byte 4	Error correction code byte 1 of the second 256 Bytes of data in this page
0x2D	ECC byte 5	Error correction code byte 2 of the second 256 Bytes of data in this page
0x2E	ECC byte 6	Error correction code byte 0 of the third 256 Bytes of data in this page
0x2F	ECC byte 7	Error correction code byte 1 of the third 256 Bytes of data in this page
0x30	ECC byte 8	Error correction code byte 2 of the third 256 Bytes of data in this page
0x31	ECC byte 9	Error correction code byte 0 of the fourth 256 Bytes of data in this page
0x32	ECC byte 10	Error correction code byte 1 of the fourth 256 Bytes of data in this page
0x33	ECC byte 11	Error correction code byte 2 of the fourth 256 Bytes of data in this page
0x34	ECC byte 12	Error correction code byte 0 of the fifth 256 Bytes of data in this page
0x35	ECC byte 13	Error correction code byte 1 of the fifth 256 Bytes of data in this page
0x36	ECC byte 14	Error correction code byte 2 of the fifth 256 Bytes of data in this page
0x37	ECC byte 15	Error correction code byte 0 of the sixth 256 Bytes of data in this page
0x38	ECC byte 16	Error correction code byte 1 of the sixth 256 Bytes of data in this page
0x39	ECC byte 17	Error correction code byte 2 of the sixth 256 Bytes of data in this page
0x3A	ECC byte 18	Error correction code byte 0 of the seventh 256 Bytes of data in this page
0x3B	ECC byte 19	Error correction code byte 1 of the seventh 256 Bytes of data in this page
0x3C	ECC byte 20	Error correction code byte 2 of the seventh 256 Bytes of data in this page
0x3D	ECC	Error correction code byte 0 of the eighth 256 Bytes of data in

35.6 Filesystem support

The NAND driver provides all necessary functions for a filesystem via the MTD interface.

Filesystems must be aware of the NAND peculiarities and restrictions. One major restrictions of NAND Flash is, that you cannot write as often as you want to a page. The consecutive writes to a page, before erasing it again, are restricted to 1-3 writes, depending on the manufacturers specifications. This applies similar to the spare area.

Therefore NAND aware filesystems must either write in page size chunks or hold a writebuffer to collect smaller writes until they sum up to pagesize. Available NAND aware filesystems: JFFS2, YAFFS.

The spare area usage to store filesystem data is controlled by the spare area placement functionality which is described in one of the earlier chapters.

35.7 Tools

The MTD project provides a couple of helpful tools to handle NAND Flash.

- flasherase, flasheraseall: Erase and format FLASH partitions
- nandwrite: write filesystem images to NAND FLASH
- nanddump: dump the contents of a NAND FLASH partitions

These tools are aware of the NAND restrictions. Please use those tools instead of complaining about errors which are caused by non NAND aware access methods.

35.8 Constants

This chapter describes the constants which might be relevant for a driver developer.

35.8.1 Chip option constants

Constants for chip id table

These constants are defined in rawnand.h. They are OR-ed together to describe the chip functionality:

```
/* Buswitdh is 16 bit */
#define NAND_BUSWIDTH_16    0x00000002
/* Device supports partial programming without padding */
#define NAND_NO_PADDING     0x00000004
/* Chip has cache program function */
#define NAND_CACHEPRG       0x00000008
/* Chip has copy back function */
#define NAND_COPYBACK       0x00000010
```

(continues on next page)

(continued from previous page)

```
/* AND Chip which has 4 banks and a confusing page / block
 * assignment. See Renesas datasheet for further information */
#define NAND_IS_AND     0x00000020
/* Chip has a array of 4 pages which can be read without
 * additional ready /busy waits */
#define NAND_4PAGE_ARRAY    0x00000040
```

Constants for runtime options

These constants are defined in rawnand.h. They are OR-ed together to describe the functionality:

```
/* The hw ecc generator provides a syndrome instead a ecc value on read
 * This can only work if we have the ecc bytes directly behind the
 * data bytes. Applies for DOC and AG-AND Renesas HW Reed Solomon␣
↪generators */
#define NAND_HWECC_SYNDROME 0x00020000
```

35.8.2 ECC selection constants

Use these constants to select the ECC algorithm:

```
/* No ECC. Usage is not recommended ! */
#define NAND_ECC_NONE       0
/* Software ECC 3 byte ECC per 256 Byte data */
#define NAND_ECC_SOFT       1
/* Hardware ECC 3 byte ECC per 256 Byte data */
#define NAND_ECC_HW3_256    2
/* Hardware ECC 3 byte ECC per 512 Byte data */
#define NAND_ECC_HW3_512    3
/* Hardware ECC 6 byte ECC per 512 Byte data */
#define NAND_ECC_HW6_512    4
/* Hardware ECC 8 byte ECC per 512 Byte data */
#define NAND_ECC_HW8_512    6
```

35.8.3 Hardware control related constants

These constants describe the requested hardware access function when the board-specific hardware control function is called:

```
/* Select the chip by setting nCE to low */
#define NAND_CTL_SETNCE     1
/* Deselect the chip by setting nCE to high */
#define NAND_CTL_CLRNCE     2
/* Select the command latch by setting CLE to high */
#define NAND_CTL_SETCLE     3
/* Deselect the command latch by setting CLE to low */
#define NAND_CTL_CLRCLE     4
/* Select the address latch by setting ALE to high */
#define NAND_CTL_SETALE     5
```

(continues on next page)

(continued from previous page)

```
/* Deselect the address latch by setting ALE to low */
#define NAND_CTL_CLRALE     6
/* Set write protection by setting WP to high. Not used! */
#define NAND_CTL_SETWP      7
/* Clear write protection by setting WP to low. Not used! */
#define NAND_CTL_CLRWP      8
```

35.8.4 Bad block table related constants

These constants describe the options used for bad block table descriptors:

```
/* Options for the bad block table descriptors */

/* The number of bits used per block in the bbt on the device */
#define NAND_BBT_NRBITS_MSK 0x0000000F
#define NAND_BBT_1BIT       0x00000001
#define NAND_BBT_2BIT       0x00000002
#define NAND_BBT_4BIT       0x00000004
#define NAND_BBT_8BIT       0x00000008
/* The bad block table is in the last good block of the device */
#define NAND_BBT_LASTBLOCK  0x00000010
/* The bbt is at the given page, else we must scan for the bbt */
#define NAND_BBT_ABSPAGE    0x00000020
/* bbt is stored per chip on multichip devices */
#define NAND_BBT_PERCHIP    0x00000080
/* bbt has a version counter at offset veroffs */
#define NAND_BBT_VERSION    0x00000100
/* Create a bbt if none axists */
#define NAND_BBT_CREATE     0x00000200
/* Write bbt if necessary */
#define NAND_BBT_WRITE      0x00001000
/* Read and write back block contents when writing bbt */
#define NAND_BBT_SAVECONTENT    0x00002000
```

35.9 Structures

This chapter contains the autogenerated documentation of the structures which are used in the NAND driver and might be relevant for a driver developer. Each struct member has a short description which is marked with an [XXX] identifier. See the chapter “Documentation hints” for an explanation.

struct **nand_parameters**
NAND generic parameters from the parameter page

Definition

```
struct nand_parameters {
  const char *model;
  bool supports_set_get_features;
  unsigned long set_feature_list[BITS_TO_LONGS(ONFI_FEATURE_NUMBER)];
  unsigned long get_feature_list[BITS_TO_LONGS(ONFI_FEATURE_NUMBER)];
```

(continues on next page)

(continued from previous page)

```
  struct onfi_params *onfi;
};
```

Members

model Model name

supports_set_get_features The NAND chip supports setting/getting features

set_feature_list Bitmap of features that can be set

get_feature_list Bitmap of features that can be get

onfi ONFI specific parameters

struct **nand_id**
NAND id structure

Definition

```
struct nand_id {
  u8 data[NAND_MAX_ID_LEN];
  int len;
};
```

Members

data buffer containing the id bytes.

len ID length.

struct **nand_ecc_step_info**
ECC step information of ECC engine

Definition

```
struct nand_ecc_step_info {
  int stepsize;
  const int *strengths;
  int nstrengths;
};
```

Members

stepsize data bytes per ECC step

strengths array of supported strengths

nstrengths number of supported strengths

struct **nand_ecc_caps**
capability of ECC engine

Definition

```
struct nand_ecc_caps {
  const struct nand_ecc_step_info *stepinfos;
  int nstepinfos;
  int (*calc_ecc_bytes)(int step_size, int strength);
};
```

Members

stepinfos array of ECC step information

nstepinfos number of ECC step information

calc_ecc_bytes driver's hook to calculate ECC bytes per step

struct **nand_ecc_ctrl**
Control structure for ECC

Definition

```
struct nand_ecc_ctrl {
  enum nand_ecc_mode mode;
  enum nand_ecc_algo algo;
  int steps;
  int size;
  int bytes;
  int total;
  int strength;
  int prepad;
  int postpad;
  unsigned int options;
  void *priv;
  u8 *calc_buf;
  u8 *code_buf;
  void (*hwctl)(struct nand_chip *chip, int mode);
  int (*calculate)(struct nand_chip *chip, const uint8_t *dat, uint8_t␣
↪*ecc_code);
  int (*correct)(struct nand_chip *chip, uint8_t *dat, uint8_t *read_ecc,␣
↪uint8_t *calc_ecc);
  int (*read_page_raw)(struct nand_chip *chip, uint8_t *buf, int oob_
↪required, int page);
  int (*write_page_raw)(struct nand_chip *chip, const uint8_t *buf, int␣
↪oob_required, int page);
  int (*read_page)(struct nand_chip *chip, uint8_t *buf, int oob_required,␣
↪int page);
  int (*read_subpage)(struct nand_chip *chip, uint32_t offs, uint32_t len,␣
↪uint8_t *buf, int page);
  int (*write_subpage)(struct nand_chip *chip, uint32_t offset,uint32_t␣
↪data_len, const uint8_t *data_buf, int oob_required, int page);
  int (*write_page)(struct nand_chip *chip, const uint8_t *buf, int oob_
↪required, int page);
  int (*write_oob_raw)(struct nand_chip *chip, int page);
  int (*read_oob_raw)(struct nand_chip *chip, int page);
  int (*read_oob)(struct nand_chip *chip, int page);
  int (*write_oob)(struct nand_chip *chip, int page);
};
```

Members

mode ECC mode

algo ECC algorithm

steps number of ECC steps per page

size data bytes per ECC step

bytes ECC bytes per step

total total number of ECC bytes per page

strength max number of correctible bits per ECC step

prepad padding information for syndrome based ECC generators

postpad padding information for syndrome based ECC generators

options ECC specific options (see NAND_ECC_XXX flags defined above)

priv pointer to private ECC control data

calc_buf buffer for calculated ECC, size is oobsize.

code_buf buffer for ECC read from flash, size is oobsize.

hwctl function to control hardware ECC generator. Must only be provided if an hardware ECC is available

calculate function for ECC calculation or readback from ECC hardware

correct function for ECC correction, matching to ECC generator (sw/hw). Should return a positive number representing the number of corrected bitflips, -EBADMSG if the number of bitflips exceed ECC strength, or any other error code if the error is not directly related to correction. If -EBADMSG is returned the input buffers should be left untouched.

read_page_raw function to read a raw page without ECC. This function should hide the specific layout used by the ECC controller and always return contiguous in-band and out-of-band data even if they' re not stored contiguously on the NAND chip (e.g. NAND_ECC_HW_SYNDROME interleaves in-band and out-of-band data).

write_page_raw function to write a raw page without ECC. This function should hide the specific layout used by the ECC controller and consider the passed data as contiguous in-band and out-of-band data. ECC controller is responsible for doing the appropriate transformations to adapt to its specific layout (e.g. NAND_ECC_HW_SYNDROME interleaves in-band and out-of-band data).

read_page function to read a page according to the ECC generator requirements; returns maximum number of bitflips corrected in any single ECC step, -EIO hw error

read_subpage function to read parts of the page covered by ECC; returns same as read_page()

write_subpage function to write parts of the page covered by ECC.

write_page function to write a page according to the ECC generator requirements.

write_oob_raw function to write chip OOB data without ECC

read_oob_raw function to read chip OOB data without ECC

read_oob function to read chip OOB data

write_oob function to write chip OOB data

struct **nand_sdr_timings**
SDR NAND chip timings

Definition

```
struct nand_sdr_timings {
  u64 tBERS_max;
  u32 tCCS_min;
  u64 tPROG_max;
  u64 tR_max;
  u32 tALH_min;
  u32 tADL_min;
  u32 tALS_min;
  u32 tAR_min;
  u32 tCEA_max;
  u32 tCEH_min;
  u32 tCH_min;
  u32 tCHZ_max;
  u32 tCLH_min;
  u32 tCLR_min;
  u32 tCLS_min;
  u32 tCOH_min;
  u32 tCS_min;
  u32 tDH_min;
  u32 tDS_min;
  u32 tFEAT_max;
  u32 tIR_min;
  u32 tITC_max;
  u32 tRC_min;
  u32 tREA_max;
  u32 tREH_min;
  u32 tRHOH_min;
  u32 tRHW_min;
  u32 tRHZ_max;
  u32 tRLOH_min;
  u32 tRP_min;
  u32 tRR_min;
  u64 tRST_max;
  u32 tWB_max;
  u32 tWC_min;
  u32 tWH_min;
  u32 tWHR_min;
  u32 tWP_min;
  u32 tWW_min;
};
```

Members

tBERS_max Block erase time

tCCS_min Change column setup time

tPROG_max Page program time

tR_max Page read time

tALH_min ALE hold time

tADL_min ALE to data loading time

tALS_min ALE setup time

tAR_min ALE to RE# delay

tCEA_max CE# access time

tCEH_min CE# high hold time

tCH_min CE# hold time

tCHZ_max CE# high to output hi-Z

tCLH_min CLE hold time

tCLR_min CLE to RE# delay

tCLS_min CLE setup time

tCOH_min CE# high to output hold

tCS_min CE# setup time

tDH_min Data hold time

tDS_min Data setup time

tFEAT_max Busy time for Set Features and Get Features

tIR_min Output hi-Z to RE# low

tITC_max Interface and Timing Mode Change time

tRC_min RE# cycle time

tREA_max RE# access time

tREH_min RE# high hold time

tRHOH_min RE# high to output hold

tRHW_min RE# high to WE# low

tRHZ_max RE# high to output hi-Z

tRLOH_min RE# low to output hold

tRP_min RE# pulse width

tRR_min Ready to RE# low (data only)

tRST_max Device reset time, measured from the falling edge of R/B# to the rising edge of R/B#.

tWB_max WE# high to SR[6] low

tWC_min WE# cycle time

tWH_min WE# high hold time

tWHR_min WE# high to RE# low

tWP_min WE# pulse width

tWW_min WP# transition to WE# low

Description

This struct defines the timing requirements of a SDR NAND chip. These information can be found in every NAND datasheets and the timings meaning are described in the ONFI specifications:

www.onfi.org/~/media/ONFI/specs/onfi_3_1_spec.pdf (chapter 4.15 Timing Parameters)

All these timings are expressed in picoseconds.

enum **nand_data_interface_type**
NAND interface timing type

Constants

NAND_SDR_IFACE Single Data Rate interface

struct **nand_data_interface**
NAND interface timing

Definition

```
struct nand_data_interface {
  enum nand_data_interface_type type;
  struct nand_timings {
    unsigned int mode;
    union {
      struct nand_sdr_timings sdr;
    };
  } timings;
};
```

Members

type type of the timing

timings The timing information

timings.mode Timing mode as defined in the specification

{unnamed_union} anonymous

timings.sdr Use it when **type** is NAND_SDR_IFACE.

const struct nand_sdr_timings * **nand_get_sdr_timings**(const struct nand_data_interface * conf)
get SDR timing from data interface

Parameters

const struct nand_data_interface * conf The data interface

struct **nand_op_cmd_instr**
Definition of a command instruction

Definition

```
struct nand_op_cmd_instr {
  u8 opcode;
};
```

Members

opcode the command to issue in one cycle

struct **nand_op_addr_instr**
Definition of an address instruction

Definition

```
struct nand_op_addr_instr {
  unsigned int naddrs;
  const u8 *addrs;
};
```

Members

naddrs length of the **addrs** array

addrs array containing the address cycles to issue

struct **nand_op_data_instr**
Definition of a data instruction

Definition

```
struct nand_op_data_instr {
  unsigned int len;
  union {
    void *in;
    const void *out;
  } buf;
  bool force_8bit;
};
```

Members

len number of data bytes to move

buf buffer to fill

buf.in buffer to fill when reading from the NAND chip

buf.out buffer to read from when writing to the NAND chip

force_8bit force 8-bit access

Description

Please note that “in” and “out” are inverted from the ONFI specification and are from the controller perspective, so a “in” is a read from the NAND chip while a “out” is a write to the NAND chip.

struct **nand_op_waitrdy_instr**
Definition of a wait ready instruction

Definition

```
struct nand_op_waitrdy_instr {
  unsigned int timeout_ms;
};
```

Members

timeout_ms maximum delay while waiting for the ready/busy pin in ms

enum **nand_op_instr_type**
Definition of all instruction types

Constants

NAND_OP_CMD_INSTR command instruction

NAND_OP_ADDR_INSTR address instruction

NAND_OP_DATA_IN_INSTR data in instruction

NAND_OP_DATA_OUT_INSTR data out instruction

NAND_OP_WAITRDY_INSTR wait ready instruction

struct **nand_op_instr**
Instruction object

Definition

```
struct nand_op_instr {
  enum nand_op_instr_type type;
  union {
    struct nand_op_cmd_instr cmd;
    struct nand_op_addr_instr addr;
    struct nand_op_data_instr data;
    struct nand_op_waitrdy_instr waitrdy;
  } ctx;
  unsigned int delay_ns;
};
```

Members

type the instruction type

ctx extra data associated to the instruction. You' ll have to use the appropriate element depending on **type**

ctx.cmd use it if **type** is NAND_OP_CMD_INSTR

ctx.addr use it if **type** is NAND_OP_ADDR_INSTR

ctx.data use it if **type** is NAND_OP_DATA_IN_INSTR or NAND_OP_DATA_OUT_INSTR

ctx.waitrdy use it if **type** is NAND_OP_WAITRDY_INSTR

delay_ns delay the controller should apply after the instruction has been issued on the bus. Most modern controllers have internal timings control logic, and in this case, the controller driver can ignore this field.

struct **nand_subop**
a sub operation

Definition

```
struct nand_subop {
  unsigned int cs;
  const struct nand_op_instr *instrs;
  unsigned int ninstrs;
  unsigned int first_instr_start_off;
  unsigned int last_instr_end_off;
};
```

Members

cs the CS line to select for this NAND sub-operation

instrs array of instructions

ninstrs length of the **instrs** array

first_instr_start_off offset to start from for the first instruction of the sub-operation

last_instr_end_off offset to end at (excluded) for the last instruction of the sub-operation

Description

Both **first_instr_start_off** and **last_instr_end_off** only apply to data or address instructions.

When an operation cannot be handled as is by the NAND controller, it will be split by the parser into sub-operations which will be passed to the controller driver.

struct **nand_op_parser_addr_constraints**
Constraints for address instructions

Definition

```
struct nand_op_parser_addr_constraints {
  unsigned int maxcycles;
};
```

Members

maxcycles maximum number of address cycles the controller can issue in a single step

struct **nand_op_parser_data_constraints**
Constraints for data instructions

Definition

```
struct nand_op_parser_data_constraints {
  unsigned int maxlen;
};
```

Members

maxlen maximum data length that the controller can handle in a single step

struct **nand_op_parser_pattern_elem**
One element of a pattern

Definition

```
struct nand_op_parser_pattern_elem {
  enum nand_op_instr_type type;
  bool optional;
  union {
    struct nand_op_parser_addr_constraints addr;
    struct nand_op_parser_data_constraints data;
```

(continues on next page)

(continued from previous page)

```
  } ctx;
};
```

Members

type the instructuction type

optional whether this element of the pattern is optional or mandatory

ctx address or data constraint

ctx.addr address constraint (number of cycles)

ctx.data data constraint (data length)

struct **nand_op_parser_pattern**
NAND sub-operation pattern descriptor

Definition

```
struct nand_op_parser_pattern {
  const struct nand_op_parser_pattern_elem *elems;
  unsigned int nelems;
  int (*exec)(struct nand_chip *chip, const struct nand_subop *subop);
};
```

Members

elems array of pattern elements

nelems number of pattern elements in **elems** array

exec the function that will issue a sub-operation

Description

A pattern is a list of elements, each element reprensenting one instruction with its constraints. The pattern itself is used by the core to match NAND chip operation with NAND controller operations. Once a match between a NAND controller operation pattern and a NAND chip operation (or a sub-set of a NAND operation) is found, the pattern ->exec() hook is called so that the controller driver can issue the operation on the bus.

Controller drivers should declare as many patterns as they support and pass this list of patterns (created with the help of the following macro) to the `nand_op_parser_exec_op()` helper.

struct **nand_op_parser**
NAND controller operation parser descriptor

Definition

```
struct nand_op_parser {
  const struct nand_op_parser_pattern *patterns;
  unsigned int npatterns;
};
```

Members

patterns array of supported patterns

npatterns length of the **patterns** array

Description

The parser descriptor is just an array of supported patterns which will be iterated by nand_op_parser_exec_op() everytime it tries to execute an NAND operation (or tries to determine if a specific operation is supported).

It is worth mentioning that patterns will be tested in their declaration order, and the first match will be taken, so it's important to order patterns appropriately so that simple/inefficient patterns are placed at the end of the list. Usually, this is where you put single instruction patterns.

struct **nand_operation**
NAND operation descriptor

Definition

```
struct nand_operation {
  unsigned int cs;
  const struct nand_op_instr *instrs;
  unsigned int ninstrs;
};
```

Members

cs the CS line to select for this NAND operation

instrs array of instructions to execute

ninstrs length of the **instrs** array

Description

The actual operation structure that will be passed to chip->exec_op().

struct **nand_controller_ops**
Controller operations

Definition

```
struct nand_controller_ops {
  int (*attach_chip)(struct nand_chip *chip);
  void (*detach_chip)(struct nand_chip *chip);
  int (*exec_op)(struct nand_chip *chip,const struct nand_operation *op,␣
↪bool check_only);
  int (*setup_data_interface)(struct nand_chip *chip, int chipnr, const␣
↪struct nand_data_interface *conf);
};
```

Members

attach_chip this method is called after the NAND detection phase after flash ID and MTD fields such as erase size, page size and OOB size have been set up. ECC requirements are available if provided by the NAND chip or device tree. Typically used to choose the appropriate ECC configuration and allocate associated resources. This hook is optional.

detach_chip free all resources allocated/claimed in nand_controller_ops->attach_chip(). This hook is optional.

exec_op controller specific method to execute NAND operations. This method replaces chip->legacy.cmdfunc(), chip->legacy.{read,write}_{buf,byte,word}(), chip->legacy.dev_ready() and chip->legacy.waifunc().

setup_data_interface setup the data interface and timing. If chipnr is set to NAND_DATA_IFACE_CHECK_ONLY this means the configuration should not be applied but only checked. This hook is optional.

struct **nand_controller**
Structure used to describe a NAND controller

Definition

```
struct nand_controller {
  struct mutex lock;
  const struct nand_controller_ops *ops;
};
```

Members

lock lock used to serialize accesses to the NAND controller

ops NAND controller operations.

struct **nand_legacy**
NAND chip legacy fields/hooks

Definition

```
struct nand_legacy {
  void __iomem *IO_ADDR_R;
  void __iomem *IO_ADDR_W;
  void (*select_chip)(struct nand_chip *chip, int cs);
  u8 (*read_byte)(struct nand_chip *chip);
  void (*write_byte)(struct nand_chip *chip, u8 byte);
  void (*write_buf)(struct nand_chip *chip, const u8 *buf, int len);
  void (*read_buf)(struct nand_chip *chip, u8 *buf, int len);
  void (*cmd_ctrl)(struct nand_chip *chip, int dat, unsigned int ctrl);
  void (*cmdfunc)(struct nand_chip *chip, unsigned command, int column,␣
↪int page_addr);
  int (*dev_ready)(struct nand_chip *chip);
  int (*waitfunc)(struct nand_chip *chip);
  int (*block_bad)(struct nand_chip *chip, loff_t ofs);
  int (*block_markbad)(struct nand_chip *chip, loff_t ofs);
  int (*set_features)(struct nand_chip *chip, int feature_addr, u8␣
↪*subfeature_para);
  int (*get_features)(struct nand_chip *chip, int feature_addr, u8␣
↪*subfeature_para);
  int chip_delay;
  struct nand_controller dummy_controller;
};
```

Members

IO_ADDR_R address to read the 8 I/O lines of the flash device

IO_ADDR_W address to write the 8 I/O lines of the flash device

select_chip select/deselect a specific target/die

read_byte read one byte from the chip

write_byte write a single byte to the chip on the low 8 I/O lines

write_buf write data from the buffer to the chip

read_buf read data from the chip into the buffer

cmd_ctrl hardware specific function for controlling ALE/CLE/nCE. Also used to write command and address

cmdfunc hardware specific function for writing commands to the chip.

dev_ready hardware specific function for accessing device ready/busy line. If set to NULL no access to ready/busy is available and the ready/busy information is read from the chip status register.

waitfunc hardware specific function for wait on ready.

block_bad check if a block is bad, using OOB markers

block_markbad mark a block bad

set_features set the NAND chip features

get_features get the NAND chip features

chip_delay chip dependent delay for transferring data from array to read regs (tR).

dummy_controller dummy controller implementation for drivers that can only control a single chip

Description

If you look at this structure you’re already wrong. These fields/hooks are all deprecated.

struct **nand_chip**
NAND Private Flash Chip Data

Definition

```
struct nand_chip {
  struct nand_device base;
  struct nand_legacy legacy;
  int (*setup_read_retry)(struct nand_chip *chip, int retry_mode);
  unsigned int options;
  unsigned int bbt_options;
  int page_shift;
  int phys_erase_shift;
  int bbt_erase_shift;
  int chip_shift;
  int pagemask;
  u8 *data_buf;
  struct {
    unsigned int bitflips;
    int page;
  } pagecache;
  int subpagesize;
  int onfi_timing_mode_default;
```

(continues on next page)

(continued from previous page)

```
  unsigned int badblockpos;
  int badblockbits;
  struct nand_id id;
  struct nand_parameters parameters;
  struct nand_data_interface data_interface;
  int cur_cs;
  int read_retries;
  struct mutex lock;
  unsigned int suspended : 1;
  int (*suspend)(struct nand_chip *chip);
  void (*resume)(struct nand_chip *chip);
  uint8_t *oob_poi;
  struct nand_controller *controller;
  struct nand_ecc_ctrl ecc;
  unsigned long buf_align;
  uint8_t *bbt;
  struct nand_bbt_descr *bbt_td;
  struct nand_bbt_descr *bbt_md;
  struct nand_bbt_descr *badblock_pattern;
  void *priv;
  struct {
    const struct nand_manufacturer *desc;
    void *priv;
  } manufacturer;
  int (*lock_area)(struct nand_chip *chip, loff_t ofs, uint64_t len);
  int (*unlock_area)(struct nand_chip *chip, loff_t ofs, uint64_t len);
};
```

Members

base Inherit from the generic NAND device

legacy All legacy fields/hooks. If you develop a new driver, don't even try to use any of these fields/hooks, and if you're modifying an existing driver that is using those fields/hooks, you should consider reworking the driver avoid using them.

setup_read_retry [FLASHSPECIFIC] flash (vendor) specific function for setting the read-retry mode. Mostly needed for MLC NAND.

options [BOARDSPECIFIC] various chip options. They can partly be set to inform nand_scan about special functionality. See the defines for further explanation.

bbt_options [INTERN] bad block specific options. All options used here must come from bbm.h. By default, these options will be copied to the appropriate nand_bbt_descr's.

page_shift [INTERN] number of address bits in a page (column address bits).

phys_erase_shift [INTERN] number of address bits in a physical eraseblock

bbt_erase_shift [INTERN] number of address bits in a bbt entry

chip_shift [INTERN] number of address bits in one chip

pagemask [INTERN] page number mask = number of (pages / chip) - 1

data_buf [INTERN] buffer for data, size is (page size + oobsize).

pagecache Structure containing page cache related fields

pagecache.bitflips Number of bitflips of the cached page

pagecache.page Page number currently in the cache. -1 means no page is currently cached

subpagesize [INTERN] holds the subpagesize

onfi_timing_mode_default [INTERN] default ONFI timing mode. This field is set to the actually used ONFI mode if the chip is ONFI compliant or deduced from the datasheet if the NAND chip is not ONFI compliant.

badblockpos [INTERN] position of the bad block marker in the oob area.

badblockbits [INTERN] minimum number of set bits in a good block's bad block marker position; i.e., BBM == 11110111b is not bad when badblockbits == 7

id [INTERN] holds NAND ID

parameters [INTERN] holds generic parameters under an easily readable form.

data_interface [INTERN] NAND interface timing information

cur_cs currently selected target. -1 means no target selected, otherwise we should always have cur_cs >= 0 && cur_cs < nanddev_ntargets(). NAND Controller drivers should not modify this value, but they're allowed to read it.

read_retries [INTERN] the number of read retry modes supported

lock lock protecting the suspended field. Also used to serialize accesses to the NAND device.

suspended set to 1 when the device is suspended, 0 when it's not.

suspend [REPLACEABLE] specific NAND device suspend operation

resume [REPLACEABLE] specific NAND device resume operation

oob_poi "poison value buffer," used for laying out OOB data before writing

controller [REPLACEABLE] a pointer to a hardware controller structure which is shared among multiple independent devices.

ecc [BOARDSPECIFIC] ECC control structure

buf_align minimum buffer alignment required by a platform

bbt [INTERN] bad block table pointer

bbt_td [REPLACEABLE] bad block table descriptor for flash lookup.

bbt_md [REPLACEABLE] bad block table mirror descriptor

badblock_pattern [REPLACEABLE] bad block scan pattern used for initial bad block scan.

priv [OPTIONAL] pointer to private chip data

manufacturer [INTERN] Contains manufacturer information

manufacturer.desc [INTERN] Contains manufacturer's description

manufacturer.priv [INTERN] Contains manufacturer private information

lock_area [REPLACEABLE] specific NAND chip lock operation

unlock_area [REPLACEABLE] specific NAND chip unlock operation

struct **nand_flash_dev**
NAND Flash Device ID Structure

Definition

```
struct nand_flash_dev {
  char *name;
  union {
    struct {
      uint8_t mfr_id;
      uint8_t dev_id;
    };
    uint8_t id[NAND_MAX_ID_LEN];
  };
  unsigned int pagesize;
  unsigned int chipsize;
  unsigned int erasesize;
  unsigned int options;
  uint16_t id_len;
  uint16_t oobsize;
  struct {
    uint16_t strength_ds;
    uint16_t step_ds;
  } ecc;
  int onfi_timing_mode_default;
};
```

Members

name a human-readable name of the NAND chip

{unnamed_union} anonymous

{unnamed_struct} anonymous

mfr_id manufacturer ID part of the full chip ID array (refers the same memory address as `id[0]`)

dev_id device ID part of the full chip ID array (refers the same memory address as `id[1]`)

id full device ID array

pagesize size of the NAND page in bytes; if 0, then the real page size (as well as the eraseblock size) is determined from the extended NAND chip ID array)

chipsize total chip size in MiB

erasesize eraseblock size in bytes (determined from the extended ID if 0)

options stores various chip bit options

id_len The valid length of the **id**.

oobsize OOB size

ecc ECC correctability and step information from the datasheet.

ecc.strength_ds The ECC correctability from the datasheet, same as the **ecc_strength_ds** in nand_chip{}.

ecc.step_ds The ECC step required by the **ecc.strength_ds**, same as the **ecc_step_ds** in nand_chip{}, also from the datasheet. For example, the "4bit ECC for each 512Byte" can be set with NAND_ECC_INFO(4, 512).

onfi_timing_mode_default the default ONFI timing mode entered after a NAND reset. Should be deduced from timings described in the datasheet.

int **nand_opcode_8bits**(unsigned int command)

Parameters

unsigned int command opcode to check

void * **nand_get_data_buf**(struct nand_chip * chip)
Get the internal page buffer

Parameters

struct nand_chip * chip NAND chip object

Description

Returns the pre-allocated page buffer after invalidating the cache. This function should be used by drivers that do not want to allocate their own bounce buffer and still need such a buffer for specific operations (most commonly when reading OOB data only).

Be careful to never call this function in the write/write_oob path, because the core may have placed the data to be written out in this buffer.

Return

pointer to the page cache buffer

35.10 Public Functions Provided

This chapter contains the autogenerated documentation of the NAND kernel API functions which are exported. Each function has a short description which is marked with an [XXX] identifier. See the chapter "Documentation hints" for an explanation.

void **nand_extract_bits**(u8 * dst, unsigned int dst_off, const u8 * src, unsigned int src_off, unsigned int nbits)
Copy unaligned bits from one buffer to another one

Parameters

u8 * dst destination buffer

unsigned int dst_off bit offset at which the writing starts

const u8 * src source buffer

unsigned int src_off bit offset at which the reading starts

unsigned int nbits number of bits to copy from **src** to **dst**

Description

Copy bits from one memory region to another (overlap authorized).

void **nand_select_target**(struct nand_chip * chip, unsigned int cs)
Select a NAND target (A.K.A. die)

Parameters

struct nand_chip * chip NAND chip object

unsigned int cs the CS line to select. Note that this CS id is always from the chip PoV, not the controller one

Description

Select a NAND target so that further operations executed on **chip** go to the selected NAND target.

void **nand_deselect_target**(struct nand_chip * chip)
Deselect the currently selected target

Parameters

struct nand_chip * chip NAND chip object

Description

Deselect the currently selected NAND target. The result of operations executed on **chip** after the target has been deselected is undefined.

int **nand_soft_waitrdy**(struct nand_chip * chip, unsigned long timeout_ms)
Poll STATUS reg until RDY bit is set to 1

Parameters

struct nand_chip * chip NAND chip structure

unsigned long timeout_ms Timeout in ms

Description

Poll the STATUS register using ->exec_op() until the RDY bit becomes 1. If that does not happen whitin the specified timeout, -ETIMEDOUT is returned.

This helper is intended to be used when the controller does not have access to the NAND R/B pin.

Be aware that calling this helper from an ->exec_op() implementation means ->exec_op() must be re-entrant.

Return 0 if the NAND chip is ready, a negative error otherwise.

int **nand_gpio_waitrdy**(struct nand_chip * chip, struct gpio_desc * gpiod,
unsigned long timeout_ms)
Poll R/B GPIO pin until ready

Parameters

struct nand_chip * chip NAND chip structure

struct gpio_desc * gpiod GPIO descriptor of R/B pin

unsigned long timeout_ms Timeout in ms

Description

Poll the R/B GPIO pin until it becomes ready. If that does not happen whitin the specified timeout, -ETIMEDOUT is returned.

This helper is intended to be used when the controller has access to the NAND R/B pin over GPIO.

Return 0 if the R/B pin indicates chip is ready, a negative error otherwise.

int **nand_read_page_op**(struct nand_chip * chip, unsigned int page, unsigned int offset_in_page, void * buf, unsigned int len)

Do a READ PAGE operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int page page to read

unsigned int offset_in_page offset within the page

void * buf buffer used to store the data

unsigned int len length of the buffer

Description

This function issues a READ PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_change_read_column_op**(struct nand_chip * chip, unsigned int offset_in_page, void * buf, unsigned int len, bool force_8bit)

Do a CHANGE READ COLUMN operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int offset_in_page offset within the page

void * buf buffer used to store the data

unsigned int len length of the buffer

bool force_8bit force 8-bit bus access

Description

This function issues a CHANGE READ COLUMN operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_read_oob_op**(struct nand_chip * chip, unsigned int page, unsigned int offset_in_oob, void * buf, unsigned int len)

Do a READ OOB operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int page page to read

unsigned int offset_in_oob offset within the OOB area

void * buf buffer used to store the data

unsigned int len length of the buffer

Description

This function issues a READ OOB operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_prog_page_begin_op**(struct nand_chip * chip, unsigned int page, unsigned int offset_in_page, const void * buf, unsigned int len)
starts a PROG PAGE operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int page page to write

unsigned int offset_in_page offset within the page

const void * buf buffer containing the data to write to the page

unsigned int len length of the buffer

Description

This function issues the first half of a PROG PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_prog_page_end_op**(struct nand_chip * chip)
ends a PROG PAGE operation

Parameters

struct nand_chip * chip The NAND chip

Description

This function issues the second half of a PROG PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_prog_page_op**(struct nand_chip * chip, unsigned int page, unsigned int offset_in_page, const void * buf, unsigned int len)
Do a full PROG PAGE operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int page page to write

unsigned int offset_in_page offset within the page

const void * buf buffer containing the data to write to the page

unsigned int len length of the buffer

Description

This function issues a full PROG PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_change_write_column_op**(struct nand_chip * chip, unsigned int offset_in_page, const void * buf, unsigned int len, bool force_8bit)

Do a CHANGE WRITE COLUMN operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int offset_in_page offset within the page

const void * buf buffer containing the data to send to the NAND

unsigned int len length of the buffer

bool force_8bit force 8-bit bus access

Description

This function issues a CHANGE WRITE COLUMN operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_readid_op**(struct nand_chip * chip, u8 addr, void * buf, unsigned int len)

Do a READID operation

Parameters

struct nand_chip * chip The NAND chip

u8 addr address cycle to pass after the READID command

void * buf buffer used to store the ID

unsigned int len length of the buffer

Description

This function sends a READID command and reads back the ID returned by the NAND. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_status_op**(struct nand_chip * chip, u8 * status)

Do a STATUS operation

Parameters

struct nand_chip * chip The NAND chip

u8 * status out variable to store the NAND status

Description

This function sends a STATUS command and reads back the status returned by the NAND. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_erase_op**(struct nand_chip * chip, unsigned int eraseblock)
Do an erase operation

Parameters

struct nand_chip * chip The NAND chip

unsigned int eraseblock block to erase

Description

This function sends an ERASE command and waits for the NAND to be ready before returning. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_reset_op**(struct nand_chip * chip)
Do a reset operation

Parameters

struct nand_chip * chip The NAND chip

Description

This function sends a RESET command and waits for the NAND to be ready before returning. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_read_data_op**(struct nand_chip * chip, void * buf, unsigned int len,
bool force_8bit, bool check_only)
Read data from the NAND

Parameters

struct nand_chip * chip The NAND chip

void * buf buffer used to store the data

unsigned int len length of the buffer

bool force_8bit force 8-bit bus access

bool check_only do not actually run the command, only checks if the controller driver supports it

Description

This function does a raw data read on the bus. Usually used after launching another NAND operation like `nand_read_page_op()`. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_write_data_op**(struct nand_chip * chip, const void * buf, unsigned int len, bool force_8bit)
Write data from the NAND

Parameters

struct nand_chip * chip The NAND chip

const void * buf buffer containing the data to send on the bus

unsigned int len length of the buffer

bool force_8bit force 8-bit bus access

Description

This function does a raw data write on the bus. Usually used after launching another NAND operation like nand_write_page_begin_op(). This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_op_parser_exec_op**(struct nand_chip * chip, const struct nand_op_parser * parser, const struct nand_operation * op, bool check_only)
exec_op parser

Parameters

struct nand_chip * chip the NAND chip

const struct nand_op_parser * parser patterns description provided by the controller driver

const struct nand_operation * op the NAND operation to address

bool check_only when true, the function only checks if **op** can be handled but does not execute the operation

Description

Helper function designed to ease integration of NAND controller drivers that only support a limited set of instruction sequences. The supported sequences are described in **parser**, and the framework takes care of splitting **op** into multiple suboperations (if required) and pass them back to the ->exec() callback of the matching pattern if **check_only** is set to false.

NAND controller drivers should call this function from their own ->exec_op() implementation.

Returns 0 on success, a negative error code otherwise. A failure can be caused by an unsupported operation (none of the supported patterns is able to handle the requested operation), or an error returned by one of the matching pattern->exec() hook.

unsigned int **nand_subop_get_addr_start_off**(const struct nand_subop * subop, unsigned int instr_idx)
Get the start offset in an address array

Parameters

const struct nand_subop * subop The entire sub-operation

unsigned int instr_idx Index of the instruction inside the sub-operation

Description

During driver development, one could be tempted to directly use the ->addr.addrs field of address instructions. This is wrong as address instructions might be split.

Given an address instruction, returns the offset of the first cycle to issue.

unsigned int **nand_subop_get_num_addr_cyc**(const struct nand_subop * subop, unsigned int instr_idx)

Get the remaining address cycles to assert

Parameters

const struct nand_subop * subop The entire sub-operation

unsigned int instr_idx Index of the instruction inside the sub-operation

Description

During driver development, one could be tempted to directly use the ->addr->naddrs field of a data instruction. This is wrong as instructions might be split.

Given an address instruction, returns the number of address cycle to issue.

unsigned int **nand_subop_get_data_start_off**(const struct nand_subop * subop, unsigned int instr_idx)

Get the start offset in a data array

Parameters

const struct nand_subop * subop The entire sub-operation

unsigned int instr_idx Index of the instruction inside the sub-operation

Description

During driver development, one could be tempted to directly use the ->data->buf.{in,out} field of data instructions. This is wrong as data instructions might be split.

Given a data instruction, returns the offset to start from.

unsigned int **nand_subop_get_data_len**(const struct nand_subop * subop, unsigned int instr_idx)

Get the number of bytes to retrieve

Parameters

const struct nand_subop * subop The entire sub-operation

unsigned int instr_idx Index of the instruction inside the sub-operation

Description

During driver development, one could be tempted to directly use the ->data->len field of a data instruction. This is wrong as data instructions might be split.

Returns the length of the chunk of data to send/receive.

int **nand_reset**(struct nand_chip * chip, int chipnr)
Reset and initialize a NAND device

Parameters

struct nand_chip * chip The NAND chip

int chipnr Internal die id

Description

Save the timings data structure, then apply SDR timings mode 0 (see nand_reset_data_interface for details), do the reset operation, and apply back the previous timings.

Returns 0 on success, a negative error code otherwise.

int **nand_check_erased_ecc_chunk**(void * data, int datalen, void * ecc, int ecclen, void * extraoob, int extraooblen, int bitflips_threshold)
check if an ECC chunk contains (almost) only 0xff data

Parameters

void * data data buffer to test

int datalen data length

void * ecc ECC buffer

int ecclen ECC length

void * extraoob extra OOB buffer

int extraooblen extra OOB length

int bitflips_threshold maximum number of bitflips

Description

Check if a data buffer and its associated ECC and OOB data contains only 0xff pattern, which means the underlying region has been erased and is ready to be programmed. The bitflips_threshold specify the maximum number of bitflips before considering the region as not erased.

Returns a positive number of bitflips less than or equal to bitflips_threshold, or -ERROR_CODE for bitflips in excess of the threshold. In case of success, the passed buffers are filled with 0xff.

Note

1/ ECC algorithms are working on pre-defined block sizes which are usually
different from the NAND page size. When fixing bitflips, ECC engines will report the number of errors per chunk, and the NAND core infrastructure expect you to return the maximum number of bitflips for the whole page. This is why you should always use this function on a single chunk and not on the whole page. After checking each chunk you should update your max_bitflips value accordingly.

2/ When checking for bitflips in erased pages you should not only check
the payload data but also their associated ECC data, because a user might

have programmed almost all bits to 1 but a few. In this case, we shouldn't consider the chunk as erased, and checking ECC bytes prevent this case.

3/ The extraoob argument is optional, and should be used if some of your OOB data are protected by the ECC engine. It could also be used if you support subpages and want to attach some extra OOB data to an ECC chunk.

int **nand_read_page_raw**(struct nand_chip * chip, uint8_t * buf, int oob_required, int page)
[INTERN] read raw page data without ecc

Parameters

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

Not for syndrome calculating ECC controllers, which use a special oob layout.

int **nand_monolithic_read_page_raw**(struct nand_chip * chip, u8 * buf, int oob_required, int page)
Monolithic page read in raw mode

Parameters

struct nand_chip * chip NAND chip info structure

u8 * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

This is a raw page read, ie. without any error detection/correction. Monolithic means we are requesting all the relevant data (main plus eventually OOB) to be loaded in the NAND cache and sent over the bus (from the NAND chip to the NAND controller) in a single operation. This is an alternative to `nand_read_page_raw()`, which first reads the main data, and if the OOB data is requested too, then reads more data on the bus.

int **nand_read_oob_std**(struct nand_chip * chip, int page)
[REPLACEABLE] the most common OOB data read function

Parameters

struct nand_chip * chip nand chip info structure

int page page number to read

int **nand_write_oob_std**(struct nand_chip * chip, int page)
[REPLACEABLE] the most common OOB data write function

Parameters

struct nand_chip * chip nand chip info structure

int page page number to write

int **nand_write_page_raw**(struct nand_chip * chip, const uint8_t * buf, int oob_required, int page)
[INTERN] raw page write function

Parameters

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

Description

Not for syndrome calculating ECC controllers, which use a special oob layout.

int **nand_monolithic_write_page_raw**(struct nand_chip * chip, const u8 * buf, int oob_required, int page)
Monolithic page write in raw mode

Parameters

struct nand_chip * chip NAND chip info structure

const u8 * buf data buffer to write

int oob_required must write chip->oob_poi to OOB

int page page number to write

Description

This is a raw page write, ie. without any error detection/correction. Monolithic means we are requesting all the relevant data (main plus eventually OOB) to be sent over the bus and effectively programmed into the NAND chip arrays in a single operation. This is an alternative to `nand_write_page_raw()`, which first sends the main data, then eventually send the OOB data by latching more data cycles on the NAND bus, and finally sends the program command to synchronyze the NAND chip cache.

int **nand_ecc_choose_conf**(struct nand_chip * chip, const struct nand_ecc_caps * caps, int oobavail)
Set the ECC strength and ECC step size

Parameters

struct nand_chip * chip nand chip info structure

const struct nand_ecc_caps * caps ECC engine caps info structure

int oobavail OOB size that the ECC engine can use

Description

Choose the ECC configuration according to following logic

1. If both ECC step size and ECC strength are already set (usually by DT) then check if it is supported by this controller.
2. If NAND_ECC_MAXIMIZE is set, then select maximum ECC strength.

3. Otherwise, try to match the ECC step size and ECC strength closest to the chip's requirement. If available OOB size can't fit the chip requirement then fallback to the maximum ECC step size and ECC strength.

On success, the chosen ECC settings are set.

int **nand_scan_with_ids**(struct nand_chip * chip, unsigned int maxchips, struct nand_flash_dev * ids)
[NAND Interface] Scan for the NAND device

Parameters

struct nand_chip * chip NAND chip object

unsigned int maxchips number of chips to scan for.

struct nand_flash_dev * ids optional flash IDs table

Description

This fills out all the uninitialized function pointers with the defaults. The flash ID is read and the mtd/chip structures are filled with the appropriate values.

void **nand_cleanup**(struct nand_chip * chip)
[NAND Interface] Free resources held by the NAND device

Parameters

struct nand_chip * chip NAND chip object

void **__nand_calculate_ecc**(const unsigned char * buf, unsigned int eccsize, unsigned char * code, bool sm_order)
[NAND Interface] Calculate 3-byte ECC for 256/512-byte block

Parameters

const unsigned char * buf input buffer with raw data

unsigned int eccsize data bytes per ECC step (256 or 512)

unsigned char * code output buffer with ECC

bool sm_order Smart Media byte ordering

int **nand_calculate_ecc**(struct nand_chip * chip, const unsigned char * buf, unsigned char * code)
[NAND Interface] Calculate 3-byte ECC for 256/512-byte block

Parameters

struct nand_chip * chip NAND chip object

const unsigned char * buf input buffer with raw data

unsigned char * code output buffer with ECC

int **__nand_correct_data**(unsigned char * buf, unsigned char * read_ecc, unsigned char * calc_ecc, unsigned int eccsize, bool sm_order)
[NAND Interface] Detect and correct bit error(s)

Parameters

unsigned char * buf raw data read from the chip

unsigned char * read_ecc ECC from the chip

unsigned char * calc_ecc the ECC calculated from raw data

unsigned int eccsize data bytes per ECC step (256 or 512)

bool sm_order Smart Media byte order

Description

Detect and correct a 1 bit error for eccsize byte block

int **nand_correct_data**(struct nand_chip * chip, unsigned char * buf, unsigned char * read_ecc, unsigned char * calc_ecc)
[NAND Interface] Detect and correct bit error(s)

Parameters

struct nand_chip * chip NAND chip object

unsigned char * buf raw data read from the chip

unsigned char * read_ecc ECC from the chip

unsigned char * calc_ecc the ECC calculated from raw data

Description

Detect and correct a 1 bit error for 256/512 byte block

35.11 Internal Functions Provided

This chapter contains the autogenerated documentation of the NAND driver internal functions. Each function has a short description which is marked with an [XXX] identifier. See the chapter "Documentation hints" for an explanation. The functions marked with [DEFAULT] might be relevant for a board driver developer.

void **nand_release_device**(struct nand_chip * chip)
[GENERIC] release chip

Parameters

struct nand_chip * chip NAND chip object

Description

Release chip lock and wake up anyone waiting on the device.

int **nand_bbm_get_next_page**(struct nand_chip * chip, int page)
Get the next page for bad block markers

Parameters

struct nand_chip * chip NAND chip object

int page First page to start checking for bad block marker usage

Description

Returns an integer that corresponds to the page offset within a block, for a page that is used to store bad block markers. If no more pages are available, -EINVAL is returned.

int **nand_block_bad**(struct nand_chip * chip, loff_t ofs)
[DEFAULT] Read bad block marker from the chip

Parameters

struct nand_chip * chip NAND chip object

loff_t ofs offset from device start

Description

Check, if the block is bad.

int **nand_get_device**(struct nand_chip * chip)
[GENERIC] Get chip for selected access

Parameters

struct nand_chip * chip NAND chip structure

Description

Lock the device and its controller for exclusive access

Return

-EBUSY if the chip has been suspended, 0 otherwise

int **nand_check_wp**(struct nand_chip * chip)
[GENERIC] check if the chip is write protected

Parameters

struct nand_chip * chip NAND chip object

Description

Check, if the device is write protected. The function expects, that the device is already selected.

uint8_t * **nand_fill_oob**(struct nand_chip * chip, uint8_t * oob, size_t len,
struct mtd_oob_ops * ops)
[INTERN] Transfer client buffer to oob

Parameters

struct nand_chip * chip NAND chip object

uint8_t * oob oob data buffer

size_t len oob data write length

struct mtd_oob_ops * ops oob ops structure

int **nand_do_write_oob**(struct nand_chip * chip, loff_t to, struct
mtd_oob_ops * ops)
[MTD Interface] NAND write out-of-band

Parameters

struct nand_chip * chip NAND chip object

loff_t to offset to write to

struct mtd_oob_ops * ops oob operation description structure

Description

NAND write out-of-band.

int **nand_default_block_markbad**(struct nand_chip * chip, loff_t ofs)
[DEFAULT] mark a block bad via bad block marker

Parameters

struct nand_chip * chip NAND chip object

loff_t ofs offset from device start

Description

This is the default implementation, which can be overridden by a hardware specific driver. It provides the details for writing a bad block marker to a block.

int **nand_markbad_bbm**(struct nand_chip * chip, loff_t ofs)
mark a block by updating the BBM

Parameters

struct nand_chip * chip NAND chip object

loff_t ofs offset of the block to mark bad

int **nand_block_markbad_lowlevel**(struct nand_chip * chip, loff_t ofs)
mark a block bad

Parameters

struct nand_chip * chip NAND chip object

loff_t ofs offset from device start

Description

This function performs the generic NAND bad block marking steps (i.e., bad block table(s) and/or marker(s)). We only allow the hardware driver to specify how to write bad block markers to OOB (chip->legacy.block_markbad).

We try operations in the following order:

(1) erase the affected block, to allow OOB marker to be written cleanly

(2) write bad block marker to OOB area of affected block (unless flag NAND_BBT_NO_OOB_BBM is present)

(3) update the BBT

Note that we retain the first error encountered in (2) or (3), finish the procedures, and dump the error in the end.

int **nand_block_isreserved**(struct mtd_info * mtd, loff_t ofs)
[GENERIC] Check if a block is marked reserved.

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset from device start

Description

Check if the block is marked as reserved.

int **nand_block_checkbad**(struct nand_chip * chip, loff_t ofs, int allowbbt)
[GENERIC] Check if a block is marked bad

Parameters

struct nand_chip * chip NAND chip object

loff_t ofs offset from device start

int allowbbt 1, if its allowed to access the bbt area

Description

Check, if the block is bad. Either by reading the bad block table or calling of the scan function.

void **panic_nand_wait**(struct nand_chip * chip, unsigned long timeo)
[GENERIC] wait until the command is done

Parameters

struct nand_chip * chip NAND chip structure

unsigned long timeo timeout

Description

Wait for command done. This is a helper function for nand_wait used when we are in interrupt context. May happen when in panic and trying to write an oops through mtdoops.

int **nand_reset_data_interface**(struct nand_chip * chip, int chipnr)
Reset data interface and timings

Parameters

struct nand_chip * chip The NAND chip

int chipnr Internal die id

Description

Reset the Data interface and timings to ONFI mode 0.

Returns 0 for success or negative error code otherwise.

int **nand_setup_data_interface**(struct nand_chip * chip, int chipnr)
Setup the best data interface and timings

Parameters

struct nand_chip * chip The NAND chip

int chipnr Internal die id

Description

Find and configure the best data interface and NAND timings supported by the chip and the driver. First tries to retrieve supported timing modes from ONFI information, and if the NAND chip does not support ONFI, relies on the ->onfi_timing_mode_default specified in the nand_ids table.

Returns 0 for success or negative error code otherwise.

int **nand_init_data_interface**(struct nand_chip * chip)
find the best data interface and timings

Parameters

struct nand_chip * chip The NAND chip

Description

Find the best data interface and NAND timings supported by the chip and the driver. First tries to retrieve supported timing modes from ONFI information, and if the NAND chip does not support ONFI, relies on the ->onfi_timing_mode_default specified in the nand_ids table. After this function nand_chip->data_interface is initialized with the best timing mode available.

Returns 0 for success or negative error code otherwise.

int **nand_fill_column_cycles**(struct nand_chip * chip, u8 * addrs, unsigned int offset_in_page)
fill the column cycles of an address

Parameters

struct nand_chip * chip The NAND chip

u8 * addrs Array of address cycles to fill

unsigned int offset_in_page The offset in the page

Description

Fills the first or the first two bytes of the **addrs** field depending on the NAND bus width and the page size.

Returns the number of cycles needed to encode the column, or a negative error code in case one of the arguments is invalid.

int **nand_read_param_page_op**(struct nand_chip * chip, u8 page, void * buf, unsigned int len)
Do a READ PARAMETER PAGE operation

Parameters

struct nand_chip * chip The NAND chip

u8 page parameter page to read

void * buf buffer used to store the data

unsigned int len length of the buffer

Description

This function issues a READ PARAMETER PAGE operation. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_exit_status_op**(struct nand_chip * chip)
Exit a STATUS operation

Parameters

struct nand_chip * chip The NAND chip

Description

This function sends a READ0 command to cancel the effect of the STATUS command to avoid reading only the status until a new read command is sent.

This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_set_features_op**(struct nand_chip * chip, u8 feature, const void * data)
Do a SET FEATURES operation

Parameters

struct nand_chip * chip The NAND chip

u8 feature feature id

const void * data 4 bytes of data

Description

This function sends a SET FEATURES command and waits for the NAND to be ready before returning. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

int **nand_get_features_op**(struct nand_chip * chip, u8 feature, void * data)
Do a GET FEATURES operation

Parameters

struct nand_chip * chip The NAND chip

u8 feature feature id

void * data 4 bytes of data

Description

This function sends a GET FEATURES command and waits for the NAND to be ready before returning. This function does not select/unselect the CS line.

Returns 0 on success, a negative error code otherwise.

struct **nand_op_parser_ctx**
Context used by the parser

Definition

```
struct nand_op_parser_ctx {
  const struct nand_op_instr *instrs;
  unsigned int ninstrs;
  struct nand_subop subop;
};
```

Members

instrs array of all the instructions that must be addressed

ninstrs length of the **instrs** array

subop Sub-operation to be passed to the NAND controller

Description

This structure is used by the core to split NAND operations into sub-operations that can be handled by the NAND controller.

bool **nand_op_parser_must_split_instr**(const struct nand_op_parser_pattern_elem * pat, const struct nand_op_instr * instr, unsigned int * start_offset)

Checks if an instruction must be split

Parameters

const struct nand_op_parser_pattern_elem * pat the parser pattern element that matches **instr**

const struct nand_op_instr * instr pointer to the instruction to check

unsigned int * start_offset this is an in/out parameter. If **instr** has already been split, then **start_offset** is the offset from which to start (either an address cycle or an offset in the data buffer). Conversely, if the function returns true (ie. instr must be split), this parameter is updated to point to the first data/address cycle that has not been taken care of.

Description

Some NAND controllers are limited and cannot send X address cycles with a unique operation, or cannot read/write more than Y bytes at the same time. In this case, split the instruction that does not fit in a single controller-operation into two or more chunks.

Returns true if the instruction must be split, false otherwise. The **start_offset** parameter is also updated to the offset at which the next bundle of instruction must start (if an address or a data instruction).

bool **nand_op_parser_match_pat**(const struct nand_op_parser_pattern * pat, struct nand_op_parser_ctx * ctx)

Checks if a pattern matches the instructions remaining in the parser context

Parameters

const struct nand_op_parser_pattern * pat the pattern to test

struct nand_op_parser_ctx * ctx the parser context structure to match with the pattern **pat**

Description

Check if **pat** matches the set or a sub-set of instructions remaining in **ctx**. Returns true if this is the case, false ortherwise. When true is returned, **ctx->subop** is updated with the set of instructions to be passed to the controller driver.

int **nand_get_features**(struct nand_chip * chip, int addr, u8 * subfeature_param)
wrapper to perform a GET_FEATURE

Parameters

struct nand_chip * chip NAND chip info structure

int addr feature address

u8 * subfeature_param the subfeature parameters, a four bytes array

Description

Returns 0 for success, a negative error otherwise. Returns -ENOTSUPP if the operation cannot be handled.

int **nand_set_features**(struct nand_chip * chip, int addr, u8 * subfeature_param)
wrapper to perform a SET_FEATURE

Parameters

struct nand_chip * chip NAND chip info structure

int addr feature address

u8 * subfeature_param the subfeature parameters, a four bytes array

Description

Returns 0 for success, a negative error otherwise. Returns -ENOTSUPP if the operation cannot be handled.

int **nand_check_erased_buf**(void * buf, int len, int bitflips_threshold)
check if a buffer contains (almost) only 0xff data

Parameters

void * buf buffer to test

int len buffer length

int bitflips_threshold maximum number of bitflips

Description

Check if a buffer contains only 0xff, which means the underlying region has been erased and is ready to be programmed. The bitflips_threshold specify the maximum number of bitflips before considering the region is not erased. Returns a positive number of bitflips less than or equal to bitflips_threshold, or -ERROR_CODE for bitflips in excess of the threshold.

Note

The logic of this function has been extracted from the memweight implementation, except that nand_check_erased_buf function exit before testing the whole buffer if the number of bitflips exceed the bitflips_threshold value.

int **nand_read_page_raw_notsupp**(struct nand_chip * chip, u8 * buf, int oob_required, int page)
dummy read raw page function

Parameters

struct nand_chip * chip nand chip info structure

u8 * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

Returns -ENOTSUPP unconditionally.

int **nand_read_page_raw_syndrome**(struct nand_chip * chip, uint8_t * buf, int oob_required, int page)
[INTERN] read raw page data without ecc

Parameters

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

We need a special oob layout and handling even when OOB isn't used.

int **nand_read_page_swecc**(struct nand_chip * chip, uint8_t * buf, int oob_required, int page)
[REPLACEABLE] software ECC based page read function

Parameters

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

int **nand_read_subpage**(struct nand_chip * chip, uint32_t data_offs, uint32_t readlen, uint8_t * bufpoi, int page)
[REPLACEABLE] ECC based sub-page read function

Parameters

struct nand_chip * chip nand chip info structure

uint32_t data_offs offset of requested data within the page

uint32_t readlen data length

uint8_t * bufpoi buffer to store read data

int page page number to read

int **nand_read_page_hwecc**(struct nand_chip * chip, uint8_t * buf, int oob_required, int page)
[REPLACEABLE] hardware ECC based page read function

Parameters

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

Not for syndrome calculating ECC controllers which need a special oob layout.

int **nand_read_page_syndrome**(struct nand_chip * chip, uint8_t * buf, int oob_required, int page)
[REPLACEABLE] hardware ECC syndrome based page read

Parameters

struct nand_chip * chip nand chip info structure

uint8_t * buf buffer to store read data

int oob_required caller requires OOB data read to chip->oob_poi

int page page number to read

Description

The hw generator calculates the error syndrome automatically. Therefore we need a special oob layout and handling.

uint8_t * **nand_transfer_oob**(struct nand_chip * chip, uint8_t * oob, struct mtd_oob_ops * ops, size_t len)
[INTERN] Transfer oob to client buffer

Parameters

struct nand_chip * chip NAND chip object

uint8_t * oob oob destination address

struct mtd_oob_ops * ops oob ops structure

size_t len size of oob to transfer

int **nand_setup_read_retry**(struct nand_chip * chip, int retry_mode)
[INTERN] Set the READ RETRY mode

Parameters

struct nand_chip * chip NAND chip object

int retry_mode the retry mode to use

Description

Some vendors supply a special command to shift the Vt threshold, to be used when there are too many bitflips in a page (i.e., ECC error). After setting a new threshold, the host should retry reading the page.

int **nand_do_read_ops**(struct nand_chip * chip, loff_t from, struct mtd_oob_ops * ops)
[INTERN] Read data with ECC

Parameters

struct nand_chip * chip NAND chip object

loff_t from offset to read from

struct mtd_oob_ops * ops oob ops structure

Description

Internal function. Called with chip held.

int **nand_read_oob_syndrome**(struct nand_chip * chip, int page)
[REPLACEABLE] OOB data read function for HW ECC with syndromes

Parameters

struct nand_chip * chip nand chip info structure

int page page number to read

int **nand_write_oob_syndrome**(struct nand_chip * chip, int page)
[REPLACEABLE] OOB data write function for HW ECC with syndrome - only for large page flash

Parameters

struct nand_chip * chip nand chip info structure

int page page number to write

int **nand_do_read_oob**(struct nand_chip * chip, loff_t from, struct mtd_oob_ops * ops)
[INTERN] NAND read out-of-band

Parameters

struct nand_chip * chip NAND chip object

loff_t from offset to read from

struct mtd_oob_ops * ops oob operations description structure

Description

NAND read out-of-band data from the spare area.

int **nand_read_oob**(struct mtd_info * mtd, loff_t from, struct mtd_oob_ops * ops)
[MTD Interface] NAND read data and/or out-of-band

Parameters

struct mtd_info * mtd MTD device structure

loff_t from offset to read from

struct mtd_oob_ops * ops oob operation description structure

Description

NAND read data and/or out-of-band data.

int **nand_write_page_raw_notsupp**(struct nand_chip * chip, const u8 * buf, int oob_required, int page)
dummy raw page write function

Parameters

struct nand_chip * chip nand chip info structure

const u8 * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

Description

Returns -ENOTSUPP unconditionally.

int **nand_write_page_raw_syndrome**(struct nand_chip * chip, const uint8_t * buf, int oob_required, int page)
[INTERN] raw page write function

Parameters

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

Description

We need a special oob layout and handling even when ECC isn't checked.

int **nand_write_page_swecc**(struct nand_chip * chip, const uint8_t * buf, int oob_required, int page)
[REPLACEABLE] software ECC based page write function

Parameters

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

int **nand_write_page_hwecc**(struct nand_chip * chip, const uint8_t * buf, int oob_required, int page)
[REPLACEABLE] hardware ECC based page write function

Parameters

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

int **nand_write_subpage_hwecc**(struct nand_chip * chip, uint32_t offset, uint32_t data_len, const uint8_t * buf, int oob_required, int page)

[REPLACEABLE] hardware ECC based subpage write

Parameters

struct nand_chip * chip nand chip info structure

uint32_t offset column address of subpage within the page

uint32_t data_len data length

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

int **nand_write_page_syndrome**(struct nand_chip * chip, const uint8_t * buf, int oob_required, int page)

[REPLACEABLE] hardware ECC syndrome based page write

Parameters

struct nand_chip * chip nand chip info structure

const uint8_t * buf data buffer

int oob_required must write chip->oob_poi to OOB

int page page number to write

Description

The hw generator calculates the error syndrome automatically. Therefore we need a special oob layout and handling.

int **nand_write_page**(struct nand_chip * chip, uint32_t offset, int data_len, const uint8_t * buf, int oob_required, int page, int raw)

write one page

Parameters

struct nand_chip * chip NAND chip descriptor

uint32_t offset address offset within the page

int data_len length of actual data to be written

const uint8_t * buf the data to write

int oob_required must write chip->oob_poi to OOB

int page page number to write

int raw use _raw version of write_page

int **nand_do_write_ops**(struct nand_chip * chip, loff_t to, struct mtd_oob_ops * ops)
[INTERN] NAND write with ECC

Parameters

struct nand_chip * chip NAND chip object

loff_t to offset to write to

struct mtd_oob_ops * ops oob operations description structure

Description

NAND write with ECC.

int **panic_nand_write**(struct mtd_info * mtd, loff_t to, size_t len, size_t * retlen, const uint8_t * buf)
[MTD Interface] NAND write with ECC

Parameters

struct mtd_info * mtd MTD device structure

loff_t to offset to write to

size_t len number of bytes to write

size_t * retlen pointer to variable to store the number of written bytes

const uint8_t * buf the data to write

Description

NAND write with ECC. Used when performing writes in interrupt context, this may for example be called by mtdoops when writing an oops while in panic.

int **nand_write_oob**(struct mtd_info * mtd, loff_t to, struct mtd_oob_ops * ops)
[MTD Interface] NAND write data and/or out-of-band

Parameters

struct mtd_info * mtd MTD device structure

loff_t to offset to write to

struct mtd_oob_ops * ops oob operation description structure

int **nand_erase**(struct mtd_info * mtd, struct erase_info * instr)
[MTD Interface] erase block(s)

Parameters

struct mtd_info * mtd MTD device structure

struct erase_info * instr erase instruction

Description

Erase one ore more blocks.

int **nand_erase_nand**(struct nand_chip * chip, struct erase_info * instr, int allowbbt)
[INTERN] erase block(s)

Parameters

struct nand_chip * chip NAND chip object

struct erase_info * instr erase instruction

int allowbbt allow erasing the bbt area

Description

Erase one ore more blocks.

void **nand_sync**(struct mtd_info * mtd)
[MTD Interface] sync

Parameters

struct mtd_info * mtd MTD device structure

Description

Sync is actually a wait for chip ready function.

int **nand_block_isbad**(struct mtd_info * mtd, loff_t offs)
[MTD Interface] Check if block at offset is bad

Parameters

struct mtd_info * mtd MTD device structure

loff_t offs offset relative to mtd start

int **nand_block_markbad**(struct mtd_info * mtd, loff_t ofs)
[MTD Interface] Mark block at the given offset as bad

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset relative to mtd start

int **nand_suspend**(struct mtd_info * mtd)
[MTD Interface] Suspend the NAND flash

Parameters

struct mtd_info * mtd MTD device structure

Description

Returns 0 for success or negative error code otherwise.

void **nand_resume**(struct mtd_info * mtd)
[MTD Interface] Resume the NAND flash

Parameters

struct mtd_info * mtd MTD device structure

void **nand_shutdown**(struct mtd_info * mtd)
[MTD Interface] Finish the current NAND operation and prevent further operations

Parameters

struct mtd_info * mtd MTD device structure

int **nand_lock**(struct mtd_info * mtd, loff_t ofs, uint64_t len)
[MTD Interface] Lock the NAND flash

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset byte address

uint64_t len number of bytes to lock (must be a multiple of block/page size)

int **nand_unlock**(struct mtd_info * mtd, loff_t ofs, uint64_t len)
[MTD Interface] Unlock the NAND flash

Parameters

struct mtd_info * mtd MTD device structure

loff_t ofs offset byte address

uint64_t len number of bytes to unlock (must be a multiple of block/page size)

int **nand_scan_ident**(struct nand_chip * chip, unsigned int maxchips, struct nand_flash_dev * table)
Scan for the NAND device

Parameters

struct nand_chip * chip NAND chip object

unsigned int maxchips number of chips to scan for

struct nand_flash_dev * table alternative NAND ID table

Description

This is the first phase of the normal nand_scan() function. It reads the flash ID and sets up MTD fields accordingly.

This helper used to be called directly from controller drivers that needed to tweak some ECC-related parameters before `nand_scan_tail()`. This separation prevented dynamic allocations during this phase which was unconvenient and as been banned for the benefit of the ->init_ecc()/cleanup_ecc() hooks.

int **nand_check_ecc_caps**(struct nand_chip * chip, const struct nand_ecc_caps * caps, int oobavail)
check the sanity of preset ECC settings

Parameters

struct nand_chip * chip nand chip info structure

const struct nand_ecc_caps * caps ECC caps info structure

int oobavail OOB size that the ECC engine can use

Description

When ECC step size and strength are already set, check if they are supported by the controller and the calculated ECC bytes fit within the chip' s OOB. On success, the calculated ECC bytes is set.

int **nand_match_ecc_req**(struct nand_chip * chip, const struct nand_ecc_caps * caps, int oobavail)
meet the chip's requirement with least ECC bytes

Parameters

struct nand_chip * chip nand chip info structure

const struct nand_ecc_caps * caps ECC engine caps info structure

int oobavail OOB size that the ECC engine can use

Description

If a chip's ECC requirement is provided, try to meet it with the least number of ECC bytes (i.e. with the largest number of OOB-free bytes). On success, the chosen ECC settings are set.

int **nand_maximize_ecc**(struct nand_chip * chip, const struct nand_ecc_caps * caps, int oobavail)
choose the max ECC strength available

Parameters

struct nand_chip * chip nand chip info structure

const struct nand_ecc_caps * caps ECC engine caps info structure

int oobavail OOB size that the ECC engine can use

Description

Choose the max ECC strength that is supported on the controller, and can fit within the chip's OOB. On success, the chosen ECC settings are set.

int **nand_scan_tail**(struct nand_chip * chip)
Scan for the NAND device

Parameters

struct nand_chip * chip NAND chip object

Description

This is the second phase of the normal nand_scan() function. It fills out all the uninitialized function pointers with the defaults and scans for a bad block table if appropriate.

int **check_pattern**(uint8_t * buf, int len, int paglen, struct nand_bbt_descr * td)
[GENERIC] check if a pattern is in the buffer

Parameters

uint8_t * buf the buffer to search

int len the length of buffer to search

int paglen the pagelength

struct nand_bbt_descr * td search pattern descriptor

Description

Check for a pattern at the given place. Used to search bad block tables and good / bad block identifiers.

int **check_short_pattern**(uint8_t * buf, struct nand_bbt_descr * td)
[GENERIC] check if a pattern is in the buffer

Parameters

uint8_t * buf the buffer to search

struct nand_bbt_descr * td search pattern descriptor

Description

Check for a pattern at the given place. Used to search bad block tables and good / bad block identifiers. Same as check_pattern, but no optional empty check.

u32 **add_marker_len**(struct nand_bbt_descr * td)
compute the length of the marker in data area

Parameters

struct nand_bbt_descr * td BBT descriptor used for computation

Description

The length will be 0 if the marker is located in OOB area.

int **read_bbt**(struct nand_chip * this, uint8_t * buf, int page, int num, struct nand_bbt_descr * td, int offs)
[GENERIC] Read the bad block table starting from page

Parameters

struct nand_chip * this NAND chip object

uint8_t * buf temporary buffer

int page the starting page

int num the number of bbt descriptors to read

struct nand_bbt_descr * td the bbt describtion table

int offs block number offset in the table

Description

Read the bad block table starting from page.

int **read_abs_bbt**(struct nand_chip * this, uint8_t * buf, struct nand_bbt_descr * td, int chip)
[GENERIC] Read the bad block table starting at a given page

Parameters

struct nand_chip * this NAND chip object

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

int chip read the table for a specific chip, -1 read all chips; applies only if NAND_BBT_PERCHIP option is set

Description

Read the bad block table for all chips starting at a given page. We assume that the bbt bits are in consecutive order.

int **scan_read_oob**(struct nand_chip * this, uint8_t * buf, loff_t offs, size_t len)
[GENERIC] Scan data+OOB region to buffer

Parameters

struct nand_chip * this NAND chip object

uint8_t * buf temporary buffer

loff_t offs offset at which to scan

size_t len length of data region to read

Description

Scan read data from data+OOB. May traverse multiple pages, interleaving page,OOB,page,OOB,⋯in buf. Completes transfer and returns the "strongest" ECC condition (error or bitflip). May quit on the first (non-ECC) error.

void **read_abs_bbts**(struct nand_chip * this, uint8_t * buf, struct nand_bbt_descr * td, struct nand_bbt_descr * md)
[GENERIC] Read the bad block table(s) for all chips starting at a given page

Parameters

struct nand_chip * this NAND chip object

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

struct nand_bbt_descr * md descriptor for the bad block table mirror

Description

Read the bad block table(s) for all chips starting at a given page. We assume that the bbt bits are in consecutive order.

int **create_bbt**(struct nand_chip * this, uint8_t * buf, struct nand_bbt_descr * bd, int chip)
[GENERIC] Create a bad block table by scanning the device

Parameters

struct nand_chip * this NAND chip object

uint8_t * buf temporary buffer

struct nand_bbt_descr * bd descriptor for the good/bad block search pattern

int chip create the table for a specific chip, -1 read all chips; applies only if NAND_BBT_PERCHIP option is set

Description

Create a bad block table by scanning the device for the given good/bad block identify pattern.

int **search_bbt**(struct nand_chip * this, uint8_t * buf, struct nand_bbt_descr * td)
[GENERIC] scan the device for a specific bad block table

Parameters

struct nand_chip * this NAND chip object

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

Description

Read the bad block table by searching for a given ident pattern. Search is preformed either from the beginning up or from the end of the device downwards. The search starts always at the start of a block. If the option NAND_BBT_PERCHIP is given, each chip is searched for a bbt, which contains the bad block information of this chip. This is necessary to provide support for certain DOC devices.

The bbt ident pattern resides in the oob area of the first page in a block.

void **search_read_bbts**(struct nand_chip * this, uint8_t * buf, struct nand_bbt_descr * td, struct nand_bbt_descr * md)
[GENERIC] scan the device for bad block table(s)

Parameters

struct nand_chip * this NAND chip object

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

struct nand_bbt_descr * md descriptor for the bad block table mirror

Description

Search and read the bad block table(s).

int **get_bbt_block**(struct nand_chip * this, struct nand_bbt_descr * td, struct nand_bbt_descr * md, int chip)
Get the first valid eraseblock suitable to store a BBT

Parameters

struct nand_chip * this the NAND device

struct nand_bbt_descr * td the BBT description

struct nand_bbt_descr * md the mirror BBT descriptor

int chip the CHIP selector

Description

This functions returns a positive block number pointing a valid eraseblock suitable to store a BBT (i.e. in the range reserved for BBT), or -ENOSPC if all blocks are already used of marked bad. If td->pages[chip] was already pointing to a valid block we re-use it, otherwise we search for the next valid one.

void **mark_bbt_block_bad**(struct nand_chip * this, struct nand_bbt_descr * td, int chip, int block)
Mark one of the block reserved for BBT bad

Parameters

struct nand_chip * this the NAND device

struct nand_bbt_descr * td the BBT description

int chip the CHIP selector

int block the BBT block to mark

Description

Blocks reserved for BBT can become bad. This functions is an helper to mark such blocks as bad. It takes care of updating the in-memory BBT, marking the block as bad using a bad block marker and invalidating the associated td->pages[] entry.

int **write_bbt**(struct nand_chip * this, uint8_t * buf, struct nand_bbt_descr * td, struct nand_bbt_descr * md, int chipsel)
[GENERIC] (Re)write the bad block table

Parameters

struct nand_chip * this NAND chip object

uint8_t * buf temporary buffer

struct nand_bbt_descr * td descriptor for the bad block table

struct nand_bbt_descr * md descriptor for the bad block table mirror

int chipsel selector for a specific chip, -1 for all

Description

(Re)write the bad block table.

int **nand_memory_bbt**(struct nand_chip * this, struct nand_bbt_descr * bd)
[GENERIC] create a memory based bad block table

Parameters

struct nand_chip * this NAND chip object

struct nand_bbt_descr * bd descriptor for the good/bad block search pattern

Description

The function creates a memory based bbt by scanning the device for manufacturer / software marked good / bad blocks.

int **check_create**(struct nand_chip * this, uint8_t * buf, struct nand_bbt_descr * bd)
[GENERIC] create and write bbt(s) if necessary

Parameters

struct nand_chip * this the NAND device

uint8_t * buf temporary buffer

struct nand_bbt_descr * bd descriptor for the good/bad block search pattern

Description

The function checks the results of the previous call to read_bbt and creates / updates the bbt(s) if necessary. Creation is necessary if no bbt was found for the chip/device. Update is necessary if one of the tables is missing or the version nr. of one table is less than the other.

int **nand_update_bbt**(struct nand_chip * this, loff_t offs)
update bad block table(s)

Parameters

struct nand_chip * this the NAND device

loff_t offs the offset of the newly marked block

Description

The function updates the bad block table(s).

void **mark_bbt_region**(struct nand_chip * this, struct nand_bbt_descr * td)
[GENERIC] mark the bad block table regions

Parameters

struct nand_chip * this the NAND device

struct nand_bbt_descr * td bad block table descriptor

Description

The bad block table regions are marked as “bad” to prevent accidental erasures / writes. The regions are identified by the mark 0x02.

void **verify_bbt_descr**(struct nand_chip * this, struct nand_bbt_descr * bd)
verify the bad block description

Parameters

struct nand_chip * this the NAND device

struct nand_bbt_descr * bd the table to verify

Description

This functions performs a few sanity checks on the bad block description table.

int **nand_scan_bbt**(struct nand_chip * this, struct nand_bbt_descr * bd)
[NAND Interface] scan, find, read and maybe create bad block table(s)

Parameters

struct nand_chip * this the NAND device

struct nand_bbt_descr * bd descriptor for the good/bad block search pattern

Description

The function checks, if a bad block table(s) is/are already available. If not it scans the device for manufacturer marked good / bad blocks and writes the bad block table(s) to the selected place.

The bad block table memory is allocated here. It must be freed by calling the nand_free_bbt function.

int **nand_create_badblock_pattern**(struct nand_chip * this)
[INTERN] Creates a BBT descriptor structure

Parameters

struct nand_chip * this NAND chip to create descriptor for

Description

This function allocates and initializes a nand_bbt_descr for BBM detection based on the properties of **this**. The new descriptor is stored in this->badblock_pattern. Thus, this->badblock_pattern should be NULL when passed to this function.

int **nand_isreserved_bbt**(struct nand_chip * this, loff_t offs)
[NAND Interface] Check if a block is reserved

Parameters

struct nand_chip * this NAND chip object

loff_t offs offset in the device

int **nand_isbad_bbt**(struct nand_chip * this, loff_t offs, int allowbbt)
[NAND Interface] Check if a block is bad

Parameters

struct nand_chip * this NAND chip object

loff_t offs offset in the device

int allowbbt allow access to bad block table region

int **nand_markbad_bbt**(struct nand_chip * this, loff_t offs)
[NAND Interface] Mark a block bad in the BBT

Parameters

struct nand_chip * this NAND chip object

loff_t offs offset of the bad block

35.12 Credits

The following people have contributed to the NAND driver:

1. Steven J. Hillsjhill@realitydiluted.com
2. David Woodhousedwmw2@infradead.org
3. Thomas Gleixnertglx@linutronix.de

A lot of users have provided bugfixes, improvements and helping hands for testing. Thanks a lot.

The following people have contributed to this document:

1. Thomas Gleixnertglx@linutronix.de

CHAPTER

THIRTYSIX

PARALLEL PORT DEVICES

int **parport_yield**(struct pardevice * dev)
relinquish a parallel port temporarily

Parameters

struct pardevice * dev a device on the parallel port

Description

This function relinquishes the port if it would be helpful to other drivers to do so. Afterwards it tries to reclaim the port using `parport_claim()`, and the return value is the same as for `parport_claim()`. If it fails, the port is left unclaimed and it is the driver's responsibility to reclaim the port.

The `parport_yield()` and `parport_yield_blocking()` functions are for marking points in the driver at which other drivers may claim the port and use their devices. Yielding the port is similar to releasing it and reclaiming it, but is more efficient because no action is taken if there are no other devices needing the port. In fact, nothing is done even if there are other devices waiting but the current device is still within its "timeslice". The default timeslice is half a second, but it can be adjusted via the /proc interface.

int **parport_yield_blocking**(struct pardevice * dev)
relinquish a parallel port temporarily

Parameters

struct pardevice * dev a device on the parallel port

Description

This function relinquishes the port if it would be helpful to other drivers to do so. Afterwards it tries to reclaim the port using `parport_claim_or_block()`, and the return value is the same as for `parport_claim_or_block()`.

int **parport_wait_event**(struct parport * port, signed long timeout)
wait for an event on a parallel port

Parameters

struct parport * port port to wait on

signed long timeout time to wait (in jiffies)

This function waits for up to **timeout** jiffies for an interrupt to occur on a parallel port. If the port timeout is set to zero, it returns immediately.

If an interrupt occurs before the timeout period elapses, this function returns zero immediately. If it times out, it returns one. An error code less than zero indicates an error (most likely a pending signal), and the calling code should finish what it's doing as soon as it can.

int **parport_wait_peripheral**(struct parport * port, unsigned char mask, unsigned char result)
wait for status lines to change in 35ms

Parameters

struct parport * port port to watch

unsigned char mask status lines to watch

unsigned char result desired values of chosen status lines

This function waits until the masked status lines have the desired values, or until 35ms have elapsed (see IEEE 1284-1994 page 24 to 25 for why this value in particular is hardcoded). The **mask** and **result** parameters are bitmasks, with the bits defined by the constants in parport.h: `PARPORT_STATUS_BUSY`, and so on.

The port is polled quickly to start off with, in anticipation of a fast response from the peripheral. This fast polling time is configurable (using /proc), and defaults to 500usec. If the timeout for this port (see `parport_set_timeout()`) is zero, the fast polling time is 35ms, and this function does not call schedule().

If the timeout for this port is non-zero, after the fast polling fails it uses `parport_wait_event()` to wait for up to 10ms, waking up if an interrupt occurs.

int **parport_negotiate**(struct parport * port, int mode)
negotiate an IEEE 1284 mode

Parameters

struct parport * port port to use

int mode mode to negotiate to

Use this to negotiate to a particular IEEE 1284 transfer mode. The **mode** parameter should be one of the constants in parport.h starting `IEEE1284_MODE_xxx`.

The return value is 0 if the peripheral has accepted the negotiation to the mode specified, -1 if the peripheral is not IEEE 1284 compliant (or not present), or 1 if the peripheral has rejected the negotiation.

ssize_t **parport_write**(struct parport * port, const void * buffer, size_t len)
write a block of data to a parallel port

Parameters

struct parport * port port to write to

const void * buffer data buffer (in kernel space)

size_t len number of bytes of data to transfer

This will write up to **len** bytes of **buffer** to the port specified, using the IEEE 1284 transfer mode most recently negotiated to (using `parport_negotiate()`), as long as that mode supports forward transfers (host to peripheral).

It is the caller's responsibility to ensure that the first **len** bytes of **buffer** are valid.

This function returns the number of bytes transferred (if zero or positive), or else an error code.

ssize_t **parport_read**(struct parport * port, void * buffer, size_t len)
read a block of data from a parallel port

Parameters

struct parport * port port to read from

void * buffer data buffer (in kernel space)

size_t len number of bytes of data to transfer

This will read up to **len** bytes of **buffer** to the port specified, using the IEEE 1284 transfer mode most recently negotiated to (using `parport_negotiate()`), as long as that mode supports reverse transfers (peripheral to host).

It is the caller's responsibility to ensure that the first **len** bytes of **buffer** are available to write to.

This function returns the number of bytes transferred (if zero or positive), or else an error code.

long **parport_set_timeout**(struct pardevice * dev, long inactivity)
set the inactivity timeout for a device

Parameters

struct pardevice * dev device on a port

long inactivity inactivity timeout (in jiffies)

This sets the inactivity timeout for a particular device on a port. This affects functions like `parport_wait_peripheral()`. The special value 0 means not to call schedule() while dealing with this device.

The return value is the previous inactivity timeout.

Any callers of `parport_wait_event()` for this device are woken up.

int **__parport_register_driver**(struct parport_driver * drv, struct module * owner, const char * mod_name)
register a parallel port device driver

Parameters

struct parport_driver * drv structure describing the driver

struct module * owner owner module of drv

const char * mod_name module name string

This can be called by a parallel port device driver in order to receive notifications about ports being found in the system, as well as ports no longer available.

If devmodel is true then the new device model is used for registration.

The **drv** structure is allocated by the caller and must not be deallocated until after calling `parport_unregister_driver()`.

If using the non device model: The driver's attach() function may block. The port that attach() is given will be valid for the duration of the callback, but if the driver wants to take a copy of the pointer it must call `parport_get_port()` to do so. Calling parport_register_device() on that port will do this for you.

The driver's detach() function may block. The port that detach() is given will be valid for the duration of the callback, but if the driver wants to take a copy of the pointer it must call `parport_get_port()` to do so.

Returns 0 on success. The non device model will always succeeds. but the new device model can fail and will return the error code.

void **parport_unregister_driver**(struct parport_driver * drv)
deregister a parallel port device driver

Parameters

struct parport_driver * drv structure describing the driver that was given to parport_register_driver()

This should be called by a parallel port device driver that has registered itself using parport_register_driver() when it is about to be unloaded.

When it returns, the driver's attach() routine will no longer be called, and for each port that attach() was called for, the detach() routine will have been called.

All the driver's attach() and detach() calls are guaranteed to have finished by the time this function returns.

struct parport * **parport_get_port**(struct parport * port)
increment a port's reference count

Parameters

struct parport * port the port

This ensures that a struct parport pointer remains valid until the matching `parport_put_port()` call.

void **parport_put_port**(struct parport * port)
decrement a port's reference count

Parameters

struct parport * port the port

This should be called once for each call to `parport_get_port()`, once the port is no longer needed. When the reference count reaches zero (port is no longer used), free_port is called.

struct parport * **parport_register_port**(unsigned long base, int irq, int dma, struct parport_operations * ops)
register a parallel port

Parameters

unsigned long base base I/O address

int irq IRQ line

int dma DMA channel

struct parport_operations * ops pointer to the port driver's port operations structure

When a parallel port (lowlevel) driver finds a port that should be made available to parallel port device drivers, it should call `parport_register_port()`. The **base**, **irq**, and **dma** parameters are for the convenience of port drivers, and for ports where they aren't meaningful needn't be set to anything special. They can be altered afterwards by adjusting the relevant members of the parport structure that is returned and represents the port. They should not be tampered with after calling parport_announce_port, however.

If there are parallel port device drivers in the system that have registered themselves using parport_register_driver(), they are not told about the port at this time; that is done by `parport_announce_port()`.

The **ops** structure is allocated by the caller, and must not be deallocated before calling `parport_remove_port()`.

If there is no memory to allocate a new parport structure, this function will return `NULL`.

void **parport_announce_port**(struct parport * port)
tell device drivers about a parallel port

Parameters

struct parport * port parallel port to announce

After a port driver has registered a parallel port with parport_register_port, and performed any necessary initialisation or adjustments, it should call `parport_announce_port()` in order to notify all device drivers that have called parport_register_driver(). Their attach() functions will be called, with **port** as the parameter.

void **parport_remove_port**(struct parport * port)
deregister a parallel port

Parameters

struct parport * port parallel port to deregister

When a parallel port driver is forcibly unloaded, or a parallel port becomes inaccessible, the port driver must call this function in order to deal with device drivers that still want to use it.

The parport structure associated with the port has its operations structure replaced with one containing 'null' operations that return errors or just don'

t do anything.

Any drivers that have registered themselves using parport_register_driver() are notified that the port is no longer accessible by having their detach() routines called with **port** as the parameter.

struct pardevice * **parport_register_dev_model**(struct parport * port, const char * name, const struct pardev_cb * par_dev_cb, int id)

register a device on a parallel port

Parameters

struct parport * port port to which the device is attached

const char * name a name to refer to the device

const struct pardev_cb * par_dev_cb struct containing callbacks

int id device number to be given to the device

This function, called by parallel port device drivers, declares that a device is connected to a port, and tells the system all it needs to know.

The struct pardev_cb contains pointer to callbacks. preemption callback function, **preempt**, is called when this device driver has claimed access to the port but another device driver wants to use it. It is given, **private**, as its parameter, and should return zero if it is willing for the system to release the port to another driver on its behalf. If it wants to keep control of the port it should return non-zero, and no action will be taken. It is good manners for the driver to try to release the port at the earliest opportunity after its preemption callback rejects a preemption attempt. Note that if a preemption callback is happy for preemption to go ahead, there is no need to release the port; it is done automatically. This function may not block, as it may be called from interrupt context. If the device driver does not support preemption, **preempt** can be `NULL`.

The wake-up (“kick”) callback function, **wakeup**, is called when the port is available to be claimed for exclusive access; that is, `parport_claim()` is guaranteed to succeed when called from inside the wake-up callback function. If the driver wants to claim the port it should do so; otherwise, it need not take any action. This function may not block, as it may be called from interrupt context. If the device driver does not want to be explicitly invited to claim the port in this way, **wakeup** can be `NULL`.

The interrupt handler, **irq_func**, is called when an interrupt arrives from the parallel port. Note that if a device driver wants to use interrupts it should use parport_enable_irq(), and can also check the irq member of the parport structure representing the port.

The parallel port (lowlevel) driver is the one that has called request_irq() and whose interrupt handler is called first. This handler does whatever needs to be done to the hardware to acknowledge the interrupt (for PC-style ports there is nothing special to be done). It then tells the IEEE 1284 code about the interrupt, which may involve reacting to an IEEE 1284 event depending

on the current IEEE 1284 phase. After this, it calls **irq_func**. Needless to say, **irq_func** will be called from interrupt context, and may not block.

The PARPORT_DEV_EXCL flag is for preventing port sharing, and so should only be used when sharing the port with other device drivers is impossible and would lead to incorrect behaviour. Use it sparingly! Normally, **flags** will be zero.

This function returns a pointer to a structure that represents the device on the port, or NULL if there is not enough memory to allocate space for that structure.

void **parport_unregister_device**(struct pardevice * dev)
deregister a device on a parallel port

Parameters

struct pardevice * dev pointer to structure representing device

This undoes the effect of parport_register_device().

struct parport * **parport_find_number**(int number)
find a parallel port by number

Parameters

int number parallel port number

This returns the parallel port with the specified number, or NULL if there is none.

There is an implicit parport_get_port() done already; to throw away the reference to the port that parport_find_number() gives you, use parport_put_port().

struct parport * **parport_find_base**(unsigned long base)
find a parallel port by base address

Parameters

unsigned long base base I/O address

This returns the parallel port with the specified base address, or NULL if there is none.

There is an implicit parport_get_port() done already; to throw away the reference to the port that parport_find_base() gives you, use parport_put_port().

int **parport_claim**(struct pardevice * dev)
claim access to a parallel port device

Parameters

struct pardevice * dev pointer to structure representing a device on the port

This function will not block and so can be used from interrupt context. If parport_claim() succeeds in claiming access to the port it returns zero and the port is available to use. It may fail (returning non-zero) if the port is in use by another driver and that driver is not willing to relinquish control of the port.

int **parport_claim_or_block**(struct pardevice * dev)
claim access to a parallel port device

Parameters

struct pardevice * dev pointer to structure representing a device on the port

This behaves like `parport_claim()`, but will block if necessary to wait for the port to be free. A return value of 1 indicates that it slept; 0 means that it succeeded without needing to sleep. A negative error code indicates failure.

void **parport_release**(struct pardevice * dev)
give up access to a parallel port device

Parameters

struct pardevice * dev pointer to structure representing parallel port device

This function cannot fail, but it should not be called without the port claimed. Similarly, if the port is already claimed you should not try claiming it again.

struct pardevice * **parport_open**(int devnum, const char * name)
find a device by canonical device number

Parameters

int devnum canonical device number

const char * name name to associate with the device

This function is similar to parport_register_device(), except that it locates a device by its number rather than by the port it is attached to.

All parameters except for **devnum** are the same as for parport_register_device(). The return value is the same as for parport_register_device().

void **parport_close**(struct pardevice * dev)
close a device opened with `parport_open()`

Parameters

struct pardevice * dev device to close

This is to `parport_open()` as `parport_unregister_device()` is to parport_register_device().

CHAPTER

THIRTYSEVEN

16X50 UART DRIVER

void **uart_update_timeout**(struct uart_port * port, unsigned int cflag, unsigned int baud)

update per-port FIFO timeout.

Parameters

struct uart_port * port uart_port structure describing the port

unsigned int cflag termios cflag value

unsigned int baud speed of the port

Set the port FIFO timeout value. The **cflag** value should reflect the actual hardware settings.

unsigned int **uart_get_baud_rate**(struct uart_port * port, struct ktermios * termios, struct ktermios * old, unsigned int min, unsigned int max)

return baud rate for a particular port

Parameters

struct uart_port * port uart_port structure describing the port in question.

struct ktermios * termios desired termios settings.

struct ktermios * old old termios (or NULL)

unsigned int min minimum acceptable baud rate

unsigned int max maximum acceptable baud rate

Decode the termios structure into a numeric baud rate, taking account of the magic 38400 baud rate (with spd_* flags), and mapping the `B0` rate to 9600 baud.

If the new baud rate is invalid, try the old termios setting. If it's still invalid, we try 9600 baud.

Update the **termios** structure to reflect the baud rate we're actually going to be using. Don't do this for the case where B0 is requested ("hang up").

unsigned int **uart_get_divisor**(struct uart_port * port, unsigned int baud)

return uart clock divisor

Parameters

struct uart_port * port uart_port structure describing the port.

unsigned int baud desired baud rate

Calculate the uart clock divisor for the port.

void **uart_console_write**(struct uart_port * port, const char * s, unsigned int count, void (*putchar)(struct uart_port *, int))

write a console message to a serial port

Parameters

struct uart_port * port the port to write the message

const char * s array of characters

unsigned int count number of characters in string to write

void (*)(struct uart_port *, int) putchar function to write character to port

int **uart_parse_earlycon**(char * p, unsigned char * iotype, resource_size_t * addr, char ** options)

Parse earlycon options

Parameters

char * p ptr to 2nd field (ie., just beyond '<name>,')

unsigned char * iotype ptr for decoded iotype (out)

resource_size_t * addr ptr for decoded mapbase/iobase (out)

char ** options ptr for <options> field; NULL if not present (out)

Decodes earlycon kernel command line parameters of the form

earlycon=<name>,io|mmio|mmio16|mmio32|mmio32be|mmio32native,<addr>,<opt
console=<name>,io|mmio|mmio16|mmio32|mmio32be|mmio32native,<addr>,<optio

The optional form

earlycon=<name>,0x<addr>,<options> console=<name>,0x<addr>,<options>

is also accepted; the returned **iotype** will be UPIO_MEM.

Returns 0 on success or -EINVAL on failure

void **uart_parse_options**(const char * options, int * baud, int * parity, int * bits, int * flow)

Parse serial port baud/parity/bits/flow control.

Parameters

const char * options pointer to option string

int * baud pointer to an 'int' variable for the baud rate.

int * parity pointer to an 'int' variable for the parity.

int * bits pointer to an 'int' variable for the number of data bits.

int * flow pointer to an 'int' variable for the flow control character.

uart_parse_options decodes a string containing the serial console options. The format of the string is <baud><parity><bits><flow>, eg: 115200n8r

int **uart_set_options**(struct uart_port * port, struct console * co, int baud, int parity, int bits, int flow)
setup the serial console parameters

Parameters

struct uart_port * port pointer to the serial ports uart_port structure

struct console * co console pointer

int baud baud rate

int parity parity character - 'n' (none), 'o' (odd), 'e' (even)

int bits number of data bits

int flow flow control character - 'r' (rts)

int **uart_register_driver**(struct uart_driver * drv)
register a driver with the uart core layer

Parameters

struct uart_driver * drv low level driver structure

Register a uart driver with the core driver. We in turn register with the tty layer, and initialise the core driver per-port state.

We have a proc file in /proc/tty/driver which is named after the normal driver.

drv->port should be NULL, and the per-port structures should be registered using uart_add_one_port after this call has succeeded.

void **uart_unregister_driver**(struct uart_driver * drv)
remove a driver from the uart core layer

Parameters

struct uart_driver * drv low level driver structure

Remove all references to a driver from the core driver. The low level driver must have removed all its ports via the `uart_remove_one_port()` if it registered them with `uart_add_one_port()`. (ie, drv->port == NULL)

int **uart_add_one_port**(struct uart_driver * drv, struct uart_port * uport)
attach a driver-defined port structure

Parameters

struct uart_driver * drv pointer to the uart low level driver structure for this port

struct uart_port * uport uart port structure to use for this port.

This allows the driver to register its own uart_port structure with the core driver. The main purpose is to allow the low level uart drivers to expand uart_port, rather than having yet more levels of structures.

int **uart_remove_one_port**(struct uart_driver * drv, struct uart_port * uport)
detach a driver defined port structure

Parameters

struct uart_driver * drv pointer to the uart low level driver structure for this port

struct uart_port * uport uart port structure for this port

This unhooks (and hangs up) the specified port structure from the core driver. No further calls will be made to the low-level code for this port.

void **uart_handle_dcd_change**(struct uart_port * uport, unsigned int status)
handle a change of carrier detect state

Parameters

struct uart_port * uport uart_port structure for the open port

unsigned int status new carrier detect status, nonzero if active

Caller must hold uport->lock

void **uart_handle_cts_change**(struct uart_port * uport, unsigned int status)
handle a change of clear-to-send state

Parameters

struct uart_port * uport uart_port structure for the open port

unsigned int status new clear to send status, nonzero if active

Caller must hold uport->lock

void **uart_insert_char**(struct uart_port * port, unsigned int status, unsigned int overrun, unsigned int ch, unsigned int flag)
push a char to the uart layer

Parameters

struct uart_port * port corresponding port

unsigned int status state of the serial port RX buffer (LSR for 8250)

unsigned int overrun mask of overrun bits in **status**

unsigned int ch character to push

unsigned int flag flag for the character (see TTY_NORMAL and friends)

Description

User is responsible to call tty_flip_buffer_push when they are done with insertion.

int **uart_get_rs485_mode**(struct uart_port * port)
retrieve rs485 properties for given uart

Parameters

struct uart_port * port undescribed

Description

This function implements the device tree binding described in Documentation/devicetree/bindings/serial/rs485.txt.

struct uart_8250_port * **serial8250_get_port**(int line)
retrieve struct uart_8250_port

Parameters

int line serial line number

Description

This function retrieves struct uart_8250_port for the specific line. This struct must not be used to perform a 8250 or serial core operation which is not accessible otherwise. Its only purpose is to make the struct accessible to the runtime-pm callbacks for context suspend/restore. The lock assumption made here is none because runtime-pm suspend/resume callbacks should not be invoked if there is any operation performed on the port.

void **serial8250_suspend_port**(int line)
suspend one serial port

Parameters

int line serial line number

Suspend one serial port.

void **serial8250_resume_port**(int line)
resume one serial port

Parameters

int line serial line number

Resume one serial port.

int **serial8250_register_8250_port**(struct uart_8250_port * up)
register a serial port

Parameters

struct uart_8250_port * up serial port template

Configure the serial port specified by the request. If the port exists and is in use, it is hung up and unregistered first.

The port is then probed and if necessary the IRQ is autodetected If this fails an error is returned.

On success the port is ready to use and the line number is returned.

void **serial8250_unregister_port**(int line)
remove a 16x50 serial port at runtime

Parameters

int line serial line number

Remove one serial port. This may not be called from interrupt context. We hand the port back to the our control.

CHAPTER

THIRTYEIGHT

PULSE-WIDTH MODULATION (PWM)

Pulse-width modulation is a modulation technique primarily used to control power supplied to electrical devices.

The PWM framework provides an abstraction for providers and consumers of PWM signals. A controller that provides one or more PWM signals is registered as `struct pwm_chip`. Providers are expected to embed this structure in a driver-specific structure. This structure contains fields that describe a particular chip.

A chip exposes one or more PWM signal sources, each of which exposed as a `struct pwm_device`. Operations can be performed on PWM devices to control the period, duty cycle, polarity and active state of the signal.

Note that PWM devices are exclusive resources: they can always only be used by one consumer at a time.

enum **pwm_polarity**
polarity of a PWM signal

Constants

PWM_POLARITY_NORMAL a high signal for the duration of the duty- cycle, followed by a low signal for the remainder of the pulse period

PWM_POLARITY_INVERSED a low signal for the duration of the duty- cycle, followed by a high signal for the remainder of the pulse period

struct **pwm_args**
board-dependent PWM arguments

Definition

```
struct pwm_args {
  unsigned int period;
  enum pwm_polarity polarity;
};
```

Members

period reference period

polarity reference polarity

Description

This structure describes board-dependent arguments attached to a PWM device. These arguments are usually retrieved from the PWM lookup table or device tree.

Do not confuse this with the PWM state: PWM arguments represent the initial configuration that users want to use on this PWM device rather than the current PWM hardware state.

struct **pwm_device**
PWM channel object

Definition

```
struct pwm_device {
  const char *label;
  unsigned long flags;
  unsigned int hwpwm;
  unsigned int pwm;
  struct pwm_chip *chip;
  void *chip_data;
  struct pwm_args args;
  struct pwm_state state;
  struct pwm_state last;
};
```

Members

label name of the PWM device

flags flags associated with the PWM device

hwpwm per-chip relative index of the PWM device

pwm global index of the PWM device

chip PWM chip providing this PWM device

chip_data chip-private data associated with the PWM device

args PWM arguments

state last applied state

last last implemented state (for PWM_DEBUG)

void **pwm_get_state**(const struct pwm_device * pwm, struct pwm_state * state)
retrieve the current PWM state

Parameters

const struct pwm_device * pwm PWM device

struct pwm_state * state state to fill with the current PWM state

void **pwm_init_state**(const struct pwm_device * pwm, struct pwm_state * state)
prepare a new state to be applied with `pwm_apply_state()`

Parameters

const struct pwm_device * pwm PWM device

struct pwm_state * state state to fill with the prepared PWM state

Description

This functions prepares a state that can later be tweaked and applied to the PWM device with `pwm_apply_state()`. This is a convenient function that first retrieves the current PWM state and the replaces the period and polarity fields with the reference values defined in pwm->args. Once the function returns, you can adjust the ->enabled and ->duty_cycle fields according to your needs before calling `pwm_apply_state()`.

->duty_cycle is initially set to zero to avoid cases where the current ->duty_cycle value exceed the pwm_args->period one, which would trigger an error if the user calls `pwm_apply_state()` without adjusting ->duty_cycle first.

unsigned int **pwm_get_relative_duty_cycle**(const struct pwm_state * state, unsigned int scale)
Get a relative duty cycle value

Parameters

const struct pwm_state * state PWM state to extract the duty cycle from

unsigned int scale target scale of the relative duty cycle

Description

This functions converts the absolute duty cycle stored in **state** (expressed in nanosecond) into a value relative to the period.

For example if you want to get the duty_cycle expressed in percent, call:

pwm_get_state(pwm, `state`); duty = pwm_get_relative_duty_cycle(`state`, 100);

int **pwm_set_relative_duty_cycle**(struct pwm_state * state, unsigned int duty_cycle, unsigned int scale)
Set a relative duty cycle value

Parameters

struct pwm_state * state PWM state to fill

unsigned int duty_cycle relative duty cycle value

unsigned int scale scale in which **duty_cycle** is expressed

Description

This functions converts a relative into an absolute duty cycle (expressed in nanoseconds), and puts the result in state->duty_cycle.

For example if you want to configure a 50% duty cycle, call:

pwm_init_state(pwm, `state`); pwm_set_relative_duty_cycle(`state`, 50, 100); pwm_apply_state(pwm, `state`);

This functions returns -EINVAL if **duty_cycle** and/or **scale** are inconsistent (**scale** == 0 or **duty_cycle** > **scale**).

struct **pwm_ops**
PWM controller operations

Definition

```
struct pwm_ops {
  int (*request)(struct pwm_chip *chip, struct pwm_device *pwm);
  void (*free)(struct pwm_chip *chip, struct pwm_device *pwm);
  int (*capture)(struct pwm_chip *chip, struct pwm_device *pwm, struct pwm_
↪capture *result, unsigned long timeout);
  int (*apply)(struct pwm_chip *chip, struct pwm_device *pwm, const struct␣
↪pwm_state *state);
  void (*get_state)(struct pwm_chip *chip, struct pwm_device *pwm, struct␣
↪pwm_state *state);
  struct module *owner;
  int (*config)(struct pwm_chip *chip, struct pwm_device *pwm, int duty_ns,
↪ int period_ns);
  int (*set_polarity)(struct pwm_chip *chip, struct pwm_device *pwm, enum␣
↪pwm_polarity polarity);
  int (*enable)(struct pwm_chip *chip, struct pwm_device *pwm);
  void (*disable)(struct pwm_chip *chip, struct pwm_device *pwm);
};
```

Members

request optional hook for requesting a PWM

free optional hook for freeing a PWM

capture capture and report PWM signal

apply atomically apply a new PWM config

get_state get the current PWM state. This function is only called once per PWM device when the PWM chip is registered.

owner helps prevent removal of modules exporting active PWMs

config configure duty cycles and period length for this PWM

set_polarity configure the polarity of this PWM

enable enable PWM output toggling

disable disable PWM output toggling

struct **pwm_chip**
abstract a PWM controller

Definition

```
struct pwm_chip {
  struct device *dev;
  const struct pwm_ops *ops;
  int base;
  unsigned int npwm;
  struct pwm_device * (*of_xlate)(struct pwm_chip *pc, const struct of_
↪phandle_args *args);
  unsigned int of_pwm_n_cells;
  struct list_head list;
  struct pwm_device *pwms;
};
```

Members

dev device providing the PWMs

ops callbacks for this PWM controller

base number of first PWM controlled by this chip

npwm number of PWMs controlled by this chip

of_xlate request a PWM device given a device tree PWM specifier

of_pwm_n_cells number of cells expected in the device tree PWM specifier

list list node for internal use

pwms array of PWM devices allocated by the framework

struct **pwm_capture**
PWM capture data

Definition

```
struct pwm_capture {
  unsigned int period;
  unsigned int duty_cycle;
};
```

Members

period period of the PWM signal (in nanoseconds)

duty_cycle duty cycle of the PWM signal (in nanoseconds)

int **pwm_config**(struct pwm_device * pwm, int duty_ns, int period_ns)
change a PWM device configuration

Parameters

struct pwm_device * pwm PWM device

int duty_ns “on” time (in nanoseconds)

int period_ns duration (in nanoseconds) of one cycle

Return

0 on success or a negative error code on failure.

int **pwm_enable**(struct pwm_device * pwm)
start a PWM output toggling

Parameters

struct pwm_device * pwm PWM device

Return

0 on success or a negative error code on failure.

void **pwm_disable**(struct pwm_device * pwm)
stop a PWM output toggling

Parameters

struct pwm_device * pwm PWM device

int **pwm_set_chip_data**(struct pwm_device * pwm, void * data)
set private chip data for a PWM

Parameters

struct pwm_device * pwm PWM device

void * data pointer to chip-specific data

Return

0 on success or a negative error code on failure.

void * **pwm_get_chip_data**(struct pwm_device * pwm)
get private chip data for a PWM

Parameters

struct pwm_device * pwm PWM device

Return

A pointer to the chip-private data for the PWM device.

int **pwmchip_add_with_polarity**(struct pwm_chip * chip, enum pwm_polarity polarity)
register a new PWM chip

Parameters

struct pwm_chip * chip the PWM chip to add

enum pwm_polarity polarity initial polarity of PWM channels

Description

Register a new PWM chip. If chip->base < 0 then a dynamically assigned base will be used. The initial polarity for all channels is specified by the **polarity** parameter.

Return

0 on success or a negative error code on failure.

int **pwmchip_add**(struct pwm_chip * chip)
register a new PWM chip

Parameters

struct pwm_chip * chip the PWM chip to add

Description

Register a new PWM chip. If chip->base < 0 then a dynamically assigned base will be used. The initial polarity for all channels is normal.

Return

0 on success or a negative error code on failure.

int **pwmchip_remove**(struct pwm_chip * chip)
remove a PWM chip

Parameters

struct pwm_chip * chip the PWM chip to remove

Description

Removes a PWM chip. This function may return busy if the PWM chip provides a PWM device that is still requested.

Return

0 on success or a negative error code on failure.

struct pwm_device * **pwm_request**(int pwm, const char * label)
request a PWM device

Parameters

int pwm global PWM device index

const char * label PWM device label

Description

This function is deprecated, use `pwm_get()` instead.

Return

A pointer to a PWM device or an ERR_PTR()-encoded error code on failure.

struct pwm_device * **pwm_request_from_chip**(struct pwm_chip * chip, unsigned int index, const char * label)
request a PWM device relative to a PWM chip

Parameters

struct pwm_chip * chip PWM chip

unsigned int index per-chip index of the PWM to request

const char * label a literal description string of this PWM

Return

A pointer to the PWM device at the given index of the given PWM chip. A negative error code is returned if the index is not valid for the specified PWM chip or if the PWM device cannot be requested.

void **pwm_free**(struct pwm_device * pwm)
free a PWM device

Parameters

struct pwm_device * pwm PWM device

Description

This function is deprecated, use `pwm_put()` instead.

int **pwm_apply_state**(struct pwm_device * pwm, const struct pwm_state * state)
atomically apply a new state to a PWM device

Parameters

struct pwm_device * pwm PWM device

const struct pwm_state * state new state to apply

int **pwm_capture**(struct pwm_device * pwm, struct pwm_capture * result, unsigned long timeout)
capture and report a PWM signal

Parameters

struct pwm_device * pwm PWM device

struct pwm_capture * result structure to fill with capture result

unsigned long timeout time to wait, in milliseconds, before giving up on capture

Return

0 on success or a negative error code on failure.

int **pwm_adjust_config**(struct pwm_device * pwm)
adjust the current PWM config to the PWM arguments

Parameters

struct pwm_device * pwm PWM device

Description

This function will adjust the PWM config to the PWM arguments provided by the DT or PWM lookup table. This is particularly useful to adapt the bootloader config to the Linux one.

struct pwm_device * **of_pwm_get**(struct device * dev, struct device_node * np, const char * con_id)
request a PWM via the PWM framework

Parameters

struct device * dev device for PWM consumer

struct device_node * np device node to get the PWM from

const char * con_id consumer name

Description

Returns the PWM device parsed from the phandle and index specified in the “pwms” property of a device tree node or a negative error-code on failure. Values parsed from the device tree are stored in the returned PWM device object.

If con_id is NULL, the first PWM device listed in the “pwms” property will be requested. Otherwise the “pwm-names” property is used to do a reverse lookup of the PWM index. This also means that the “pwm-names” property becomes mandatory for devices that look up the PWM device via the con_id parameter.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

struct pwm_device * **pwm_get**(struct device * dev, const char * con_id)
look up and request a PWM device

Parameters

struct device * dev device for PWM consumer

const char * con_id consumer name

Description

Lookup is first attempted using DT. If the device was not instantiated from a device tree, a PWM chip and a relative index is looked up via a table supplied by board setup code (see pwm_add_table()).

Once a PWM chip has been found the specified PWM device will be requested and is ready to be used.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

void **pwm_put**(struct pwm_device * pwm)
release a PWM device

Parameters

struct pwm_device * pwm PWM device

struct pwm_device * **devm_pwm_get**(struct device * dev, const char * con_id)
resource managed `pwm_get()`

Parameters

struct device * dev device for PWM consumer

const char * con_id consumer name

Description

This function performs like `pwm_get()` but the acquired PWM device will automatically be released on driver detach.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

struct pwm_device * **devm_of_pwm_get**(struct device * dev, struct device_node * np, const char * con_id)
resource managed `of_pwm_get()`

Parameters

struct device * dev device for PWM consumer

struct device_node * np device node to get the PWM from

const char * con_id consumer name

Description

This function performs like `of_pwm_get()` but the acquired PWM device will automatically be released on driver detach.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

struct pwm_device * **devm_fwnode_pwm_get**(struct device * dev, struct fwnode_handle * fwnode, const char * con_id)
request a resource managed PWM from firmware node

Parameters

struct device * dev device for PWM consumer

struct fwnode_handle * fwnode firmware node to get the PWM from

const char * con_id consumer name

Description

Returns the PWM device parsed from the firmware node. See `of_pwm_get()` and acpi_pwm_get() for a detailed description.

Return

A pointer to the requested PWM device or an ERR_PTR()-encoded error code on failure.

void **devm_pwm_put**(struct device * dev, struct pwm_device * pwm)
resource managed `pwm_put()`

Parameters

struct device * dev device for PWM consumer

struct pwm_device * pwm PWM device

Description

Release a PWM previously allocated using `devm_pwm_get()`. Calling this function is usually not needed because devm-allocated resources are automatically released on driver detach.

CHAPTER

THIRTYNINE

INTEL(R) MANAGEMENT ENGINE INTERFACE (INTEL(R) MEI)

39.1 Introduction

The Intel Management Engine (Intel ME) is an isolated and protected computing resource (Co-processor) residing inside certain Intel chipsets. The Intel ME provides support for computer/IT management and security features. The actual feature set depends on the Intel chipset SKU.

The Intel Management Engine Interface (Intel MEI, previously known as HECI) is the interface between the Host and Intel ME. This interface is exposed to the host as a PCI device, actually multiple PCI devices might be exposed. The Intel MEI Driver is in charge of the communication channel between a host application and the Intel ME features.

Each Intel ME feature, or Intel ME Client is addressed by a unique GUID and each client has its own protocol. The protocol is message-based with a header and payload up to maximal number of bytes advertised by the client, upon connection.

39.2 Intel MEI Driver

The driver exposes a character device with device nodes /dev/meiX.

An application maintains communication with an Intel ME feature while /dev/meiX is open. The binding to a specific feature is performed by calling `MEI_CONNECT_CLIENT_IOCTL`, which passes the desired GUID. The number of instances of an Intel ME feature that can be opened at the same time depends on the Intel ME feature, but most of the features allow only a single instance.

The driver is transparent to data that are passed between firmware feature and host application.

Because some of the Intel ME features can change the system configuration, the driver by default allows only a privileged user to access it.

The session is terminated calling `close(int fd)()`.

A code snippet for an application communicating with Intel AMTHI client:

```
struct mei_connect_client_data data;
fd = open(MEI_DEVICE);

data.d.in_client_uuid = AMTHI_GUID;

ioctl(fd, IOCTL_MEI_CONNECT_CLIENT, &data);

printf("Ver=%d, MaxLen=%ld\n",
       data.d.in_client_uuid.protocol_version,
       data.d.in_client_uuid.max_msg_length);

[...]

write(fd, amthi_req_data, amthi_req_data_len);

[...]

read(fd, &amthi_res_data, amthi_res_data_len);

[...]
close(fd);
```

User space API

39.3 IOCTLs:

The Intel MEI Driver supports the following IOCTL commands:

39.3.1 IOCTL_MEI_CONNECT_CLIENT

Connect to firmware Feature/Client.

```
Usage:

struct mei_connect_client_data client_data;

ioctl(fd, IOCTL_MEI_CONNECT_CLIENT, &client_data);

Inputs:

struct mei_connect_client_data - contain the following
Input field:

        in_client_uuid -        GUID of the FW Feature that needs
                                to connect to.
 Outputs:
        out_client_properties - Client Properties: MTU and Protocol␣
↪Version.

 Error returns:

        ENOTTY  No such client (i.e. wrong GUID) or connection is not␣
↪allowed.
```

(continues on next page)

(continued from previous page)

```
        EINVAL  Wrong IOCTL Number
        ENODEV  Device or Connection is not initialized or ready.
        ENOMEM  Unable to allocate memory to client internal data.
        EFAULT  Fatal Error (e.g. Unable to access user input data)
        EBUSY   Connection Already Open
```

Note max_msg_length (MTU) in client properties describes the maximum data that can be sent or received. (e.g. if MTU=2K, can send requests up to bytes 2k and received responses up to 2k bytes).

39.3.2 IOCTL_MEI_NOTIFY_SET

Enable or disable event notifications.

```
Usage:

        uint32_t enable;

        ioctl(fd, IOCTL_MEI_NOTIFY_SET, &enable);


        uint32_t enable = 1;
        or
        uint32_t enable[disable] = 0;

Error returns:


        EINVAL  Wrong IOCTL Number
        ENODEV  Device  is not initialized or the client not connected
        ENOMEM  Unable to allocate memory to client internal data.
        EFAULT  Fatal Error (e.g. Unable to access user input data)
        EOPNOTSUPP if the device doesn't support the feature
```

Note The client must be connected in order to enable notification events

39.3.3 IOCTL_MEI_NOTIFY_GET

Retrieve event

```
Usage:
        uint32_t event;
        ioctl(fd, IOCTL_MEI_NOTIFY_GET, &event);

Outputs:
        1 - if an event is pending
        0 - if there is no even pending

Error returns:
        EINVAL  Wrong IOCTL Number
        ENODEV  Device is not initialized or the client not connected
        ENOMEM  Unable to allocate memory to client internal data.
```

(continues on next page)

(continued from previous page)

```
        EFAULT  Fatal Error (e.g. Unable to access user input data)
        EOPNOTSUPP if the device doesn't support the feature
```

Note The client must be connected and event notification has to be enabled in order to receive an event

39.4 Supported Chipsets

82X38/X48 Express and newer

linux-mei@linux.intel.com

39.5 Intel(R) Management Engine (ME) Client bus API

39.5.1 Rationale

The MEI character device is useful for dedicated applications to send and receive data to the many FW appliance found in Intel' s ME from the user space. However, for some of the ME functionalities it makes sense to leverage existing software stack and expose them through existing kernel subsystems.

In order to plug seamlessly into the kernel device driver model we add kernel virtual bus abstraction on top of the MEI driver. This allows implementing Linux kernel drivers for the various MEI features as a stand alone entities found in their respective subsystem. Existing device drivers can even potentially be re-used by adding an MEI CL bus layer to the existing code.

39.5.2 MEI CL bus API

A driver implementation for an MEI Client is very similar to any other existing bus based device drivers. The driver registers itself as an MEI CL bus driver through the `struct mei_cl_driver` structure defined in `include/linux/mei_cl_bus.c`

```
struct mei_cl_driver {
        struct device_driver driver;
        const char *name;

        const struct mei_cl_device_id *id_table;

        int (*probe)(struct mei_cl_device *dev, const struct mei_cl_id␣
↪*id);
        int (*remove)(struct mei_cl_device *dev);
};
```

The mei_cl_device_id structure defined in `include/linux/mod_devicetable.h` allows a driver to bind itself against a device name.

```
struct mei_cl_device_id {
        char name[MEI_CL_NAME_SIZE];
        uuid_le uuid;
        __u8    version;
        kernel_ulong_t driver_info;
};
```

To actually register a driver on the ME Client bus one must call the `mei_cl_add_driver()` API. This is typically called at module initialization time.

Once the driver is registered and bound to the device, a driver will typically try to do some I/O on this bus and this should be done through the `mei_cl_send()` and `mei_cl_recv()` functions. More detailed information is in API: section.

In order for a driver to be notified about pending traffic or event, the driver should register a callback via `mei_cl_devev_register_rx_cb()` and `mei_cldev_register_notify_cb()` function respectively.

API:

ssize_t **mei_cldev_send**(struct mei_cl_device * cldev, u8 * buf, size_t length)
me device send (write)

Parameters

struct mei_cl_device * cldev me client device

u8 * buf buffer to send

size_t length buffer length

Return

written size in bytes or < 0 on error

ssize_t **mei_cldev_recv_nonblock**(struct mei_cl_device * cldev, u8 * buf, size_t length)
non block client receive (read)

Parameters

struct mei_cl_device * cldev me client device

u8 * buf buffer to receive

size_t length buffer length

Return

read size in bytes of < 0 on error -EAGAIN if function will block.

ssize_t **mei_cldev_recv**(struct mei_cl_device * cldev, u8 * buf, size_t length)
client receive (read)

Parameters

struct mei_cl_device * cldev me client device

u8 * buf buffer to receive

size_t length buffer length

Return

read size in bytes of < 0 on error

int **mei_cldev_register_rx_cb**(struct mei_cl_device * cldev, mei_cldev_cb_t rx_cb)
register Rx event callback

Parameters

struct mei_cl_device * cldev me client devices

mei_cldev_cb_t rx_cb callback function

Return

0 on success -EALREADY if an callback is already registered <0 on other errors

int **mei_cldev_register_notif_cb**(struct mei_cl_device * cldev, mei_cldev_cb_t notif_cb)
register FW notification event callback

Parameters

struct mei_cl_device * cldev me client devices

mei_cldev_cb_t notif_cb callback function

Return

0 on success -EALREADY if an callback is already registered <0 on other errors

void * **mei_cldev_get_drvdata**(const struct mei_cl_device * cldev)
driver data getter

Parameters

const struct mei_cl_device * cldev mei client device

Return

driver private data

void **mei_cldev_set_drvdata**(struct mei_cl_device * cldev, void * data)
driver data setter

Parameters

struct mei_cl_device * cldev mei client device

void * data data to store

const uuid_le * **mei_cldev_uuid**(const struct mei_cl_device * cldev)
return uuid of the underlying me client

Parameters

const struct mei_cl_device * cldev mei client device

Return

me client uuid

u8 **mei_cldev_ver**(const struct mei_cl_device * cldev)
return protocol version of the underlying me client

Parameters

const struct mei_cl_device * cldev mei client device

Return

me client protocol version

bool **mei_cldev_enabled**(struct mei_cl_device * cldev)
check whether the device is enabled

Parameters

struct mei_cl_device * cldev mei client device

Return

true if me client is initialized and connected

int **mei_cldev_enable**(struct mei_cl_device * cldev)
enable me client device create connection with me client

Parameters

struct mei_cl_device * cldev me client device

Return

0 on success and < 0 on error

int **mei_cldev_disable**(struct mei_cl_device * cldev)
disable me client device disconnect form the me client

Parameters

struct mei_cl_device * cldev me client device

Return

0 on success and < 0 on error

39.5.3 Example

As a theoretical example let's pretend the ME comes with a "contact" NFC IP. The driver init and exit routines for this device would look like:

```
#define CONTACT_DRIVER_NAME "contact"

static struct mei_cl_device_id contact_mei_cl_tbl[] = {
        { CONTACT_DRIVER_NAME, },

        /* required last entry */
        { }
};
MODULE_DEVICE_TABLE(mei_cl, contact_mei_cl_tbl);

static struct mei_cl_driver contact_driver = {
        .id_table = contact_mei_tbl,
        .name = CONTACT_DRIVER_NAME,

        .probe = contact_probe,
```

(continues on next page)

(continued from previous page)

```
        .remove = contact_remove,
};

static int contact_init(void)
{
        int r;

        r = mei_cl_driver_register(&contact_driver);
        if (r) {
                pr_err(CONTACT_DRIVER_NAME ": driver registration failed\n
↪");
                return r;
        }

        return 0;
}

static void __exit contact_exit(void)
{
        mei_cl_driver_unregister(&contact_driver);
}

module_init(contact_init);
module_exit(contact_exit);
```

And the driver's simplified probe routine would look like that:

```
int contact_probe(struct mei_cl_device *dev, struct mei_cl_device_id *id)
{
        [...]
        mei_cldev_enable(dev);

        mei_cldev_register_rx_cb(dev, contact_rx_cb);

        return 0;
}
```

In the probe routine the driver first enable the MEI device and then registers an rx handler which is as close as it can get to registering a threaded IRQ handler. The handler implementation will typically call `mei_cldev_recv()` and then process received data.

```
#define MAX_PAYLOAD 128
#define HDR_SIZE 4
static void conntact_rx_cb(struct mei_cl_device *cldev)
{
        struct contact *c = mei_cldev_get_drvdata(cldev);
        unsigned char payload[MAX_PAYLOAD];
        ssize_t payload_sz;

        payload_sz = mei_cldev_recv(cldev, payload,  MAX_PAYLOAD)
        if (reply_size < HDR_SIZE) {
                return;
        }
```

(continues on next page)

(continued from previous page)

```
        c->process_rx(payload);

}
```

39.5.4 MEI Client Bus Drivers

HDCP:

ME FW as a security engine provides the capability for setting up HDCP2.2 protocol negotiation between the Intel graphics device and an HDC2.2 sink.

ME FW prepares HDCP2.2 negotiation parameters, signs and encrypts them according the HDCP 2.2 spec. The Intel graphics sends the created blob to the HDCP2.2 sink.

Similarly, the HDCP2.2 sink's response is transferred to ME FW for decryption and verification.

Once all the steps of HDCP2.2 negotiation are completed, upon request ME FW will configure the port as authenticated and supply the HDCP encryption keys to Intel graphics hardware.

mei_hdcp driver

The mei_hdcp driver acts as a translation layer between HDCP 2.2 protocol implementer (I915) and ME FW by translating HDCP2.2 negotiation messages to ME FW command payloads and vice versa.

mei_hdcp api

int **mei_hdcp_initiate_session**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_ake_init * ake_data)

Initiate a Wired HDCP2.2 Tx Session in ME FW

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_ake_init * ake_data AKE_Init msg output.

Return

0 on Success, <0 on Failure.

int **mei_hdcp_verify_receiver_cert_prepare_km**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_ake_send_cert * rx_cert, bool * km_stored, struct hdcp2_ake_no_stored_km * ek_pub_km, size_t * msg_sz)

Verify the Receiver Certificate AKE_Send_Cert and prepare AKE_Stored_Km/AKE_No_Stored_Km

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_ake_send_cert * rx_cert AKE_Send_Cert for verification

bool * km_stored Pairing status flag output

struct hdcp2_ake_no_stored_km * ek_pub_km AKE_Stored_Km/AKE_No_Stored_Km output msg

size_t * msg_sz size of AKE_XXXXX_Km output msg

Return

0 on Success, <0 on Failure

int **mei_hdcp_verify_hprime**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_ake_send_hprime * rx_hprime)

Verify AKE_Send_H_prime at ME FW.

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_ake_send_hprime * rx_hprime AKE_Send_H_prime msg for ME FW verification

Return

0 on Success, <0 on Failure

int **mei_hdcp_store_pairing_info**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_ake_send_pairing_info * pairing_info)

Store pairing info received at ME FW

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_ake_send_pairing_info * pairing_info
AKE_Send_Pairing_Info msg input to ME FW

Return

0 on Success, <0 on Failure

int **mei_hdcp_initiate_locality_check**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_lc_init * lc_init_data)
Prepare LC_Init

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_lc_init * lc_init_data LC_Init msg output

Return

0 on Success, <0 on Failure

int **mei_hdcp_verify_lprime**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_lc_send_lprime * rx_lprime)
Verify lprime.

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_lc_send_lprime * rx_lprime LC_Send_L_prime msg for ME FW verification

Return

0 on Success, <0 on Failure

int **mei_hdcp_get_session_key**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_ske_send_eks * ske_data)
Prepare SKE_Send_Eks.

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_ske_send_eks * ske_data SKE_Send_Eks msg output from ME FW.

Return

0 on Success, <0 on Failure

int **mei_hdcp_repeater_check_flow_prepare_ack**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_rep_send_receiverid_list * rep_topology, struct hdcp2_rep_send_ack * rep_send_ack)

Validate the Downstream topology and prepare rep_ack.

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_rep_send_receiverid_list * rep_topology Receiver ID List to be validated

struct hdcp2_rep_send_ack * rep_send_ack repeater ack from ME FW.

Return

0 on Success, <0 on Failure

int **mei_hdcp_verify_mprime**(struct device * dev, struct hdcp_port_data * data, struct hdcp2_rep_stream_ready * stream_ready)

Verify mprime.

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

struct hdcp2_rep_stream_ready * stream_ready RepeaterAuth_Stream_Ready msg for ME FW verification.

Return

0 on Success, <0 on Failure

int **mei_hdcp_enable_authentication**(struct device * dev, struct hdcp_port_data * data)

Mark a port as authenticated through ME FW

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

Return

0 on Success, <0 on Failure

int **mei_hdcp_close_session**(struct device * dev, struct hdcp_port_data * data)

Close the Wired HDCP Tx session of ME FW per port. This also disables the authenticated state of the port.

Parameters

struct device * dev device corresponding to the mei_cl_device

struct hdcp_port_data * data Intel HW specific hdcp data

Return

0 on Success, <0 on Failure

int **mei_hdcp_component_match**(struct device * dev, int subcomponent, void * data)
compare function for matching mei hdcp.

Parameters

struct device * dev master device

int subcomponent subcomponent to match (I915_COMPONENT_HDCP)

void * data compare data (mei hdcp device)

Description

The function checks if the driver is i915, the subcomponent is HDCP and the grand parent of hdcp and the parent of i915 are the same PCH device.

Return

- 1 - if components match
- 0 - otherwise

MEI NFC

Some Intel 8 and 9 Serieses chipsets supports NFC devices connected behind the Intel Management Engine controller. MEI client bus exposes the NFC chips as NFC phy devices and enables binding with Microread and NXP PN544 NFC device driver from the Linux NFC subsystem.

39.6 Intel(R) Active Management Technology (Intel AMT)

Prominent usage of the Intel ME Interface is to communicate with Intel(R) Active Management Technology (Intel AMT) implemented in firmware running on the Intel ME.

Intel AMT provides the ability to manage a host remotely out-of-band (OOB) even when the operating system running on the host processor has crashed or is in a sleep state.

Some examples of Intel AMT usage are:

- Monitoring hardware state and platform components
- Remote power off/on (useful for green computing or overnight IT maintenance)
- OS updates

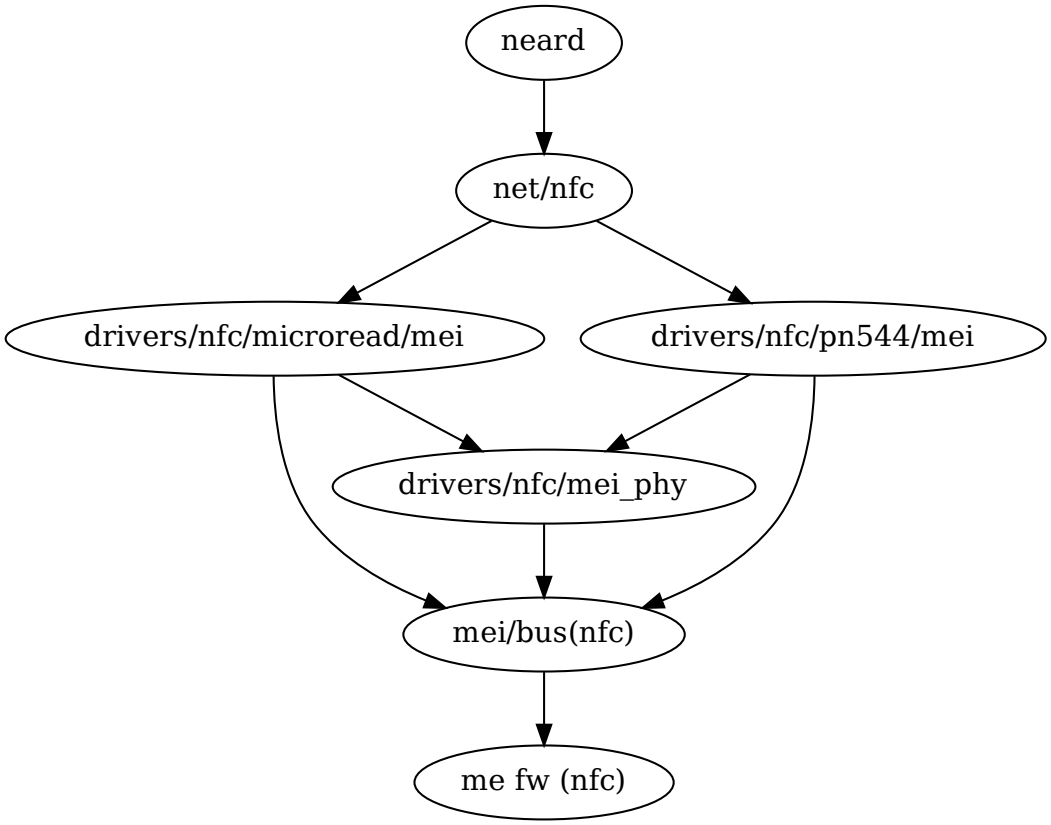


Fig. 1: **MEI NFC** Stack

- Storage of useful platform information such as software assets
- Built-in hardware KVM
- Selective network isolation of Ethernet and IP protocol flows based on policies set by a remote management console
- IDE device redirection from remote management console

Intel AMT (OOB) communication is based on SOAP (deprecated starting with Release 6.0) over HTTP/S or WS-Management protocol over HTTP/S that are received from a remote management console application.

For more information about Intel AMT: https://software.intel.com/sites/manageability/AMT_Implementation_and_Reference_Guide/default.htm

39.6.1 Intel AMT Applications

1) Intel Local Management Service (Intel LMS)

   Applications running locally on the platform communicate with Intel AMT Release 2.0 and later releases in the same way that network applications do via SOAP over HTTP (deprecated starting with Release 6.0) or with WS-Management over SOAP over HTTP. This means that some Intel AMT features can be accessed from a local application using the same network interface as a remote application communicating with Intel AMT over the network.

   When a local application sends a message addressed to the local Intel AMT host name, the Intel LMS, which listens for traffic directed to the host name, intercepts the message and routes it to the Intel MEI. For more information: https://software.intel.com/sites/manageability/AMT_Implementation_and_Reference_Guide/default.htm Under "About Intel AMT" => "Local Access"

   For downloading Intel LMS: https://github.com/intel/lms

   The Intel LMS opens a connection using the Intel MEI driver to the Intel LMS firmware feature using a defined GUID and then communicates with the feature using a protocol called Intel AMT Port Forwarding Protocol (Intel APF protocol). The protocol is used to maintain multiple sessions with Intel AMT from a single application.

   See the protocol specification in the Intel AMT Software Development Kit (SDK) https://software.intel.com/sites/manageability/AMT_Implementation_and_Reference_Guide/default.htm Under "SDK Resources" => "Intel(R) vPro(TM) Gateway (MPS)" => "Information for Intel(R) vPro(TM) Gateway Developers" => "Description of the Intel AMT Port Forwarding (APF) Protocol"

2) Intel AMT Remote configuration using a Local Agent

   A Local Agent enables IT personnel to configure Intel AMT out-of-the-box without requiring installing additional data to enable setup. The remote configuration process may involve an ISV-developed remote configuration agent that runs on the host. For more information: https://software.intel.com/sites/

manageability/AMT_Implementation_and_Reference_Guide/default.htm Under “Setup and Configuration of Intel AMT” => “SDK Tools Supporting Setup and Configuration” => “Using the Local Agent Sample”

39.6.2 Intel AMT OS Health Watchdog

The Intel AMT Watchdog is an OS Health (Hang/Crash) watchdog. Whenever the OS hangs or crashes, Intel AMT will send an event to any subscriber to this event. This mechanism means that IT knows when a platform crashes even when there is a hard failure on the host.

The Intel AMT Watchdog is composed of two parts:

1) Firmware feature - receives the heartbeats and sends an event when the heartbeats stop.
2) Intel MEI iAMT watchdog driver - connects to the watchdog feature, configures the watchdog and sends the heartbeats.

The Intel iAMT watchdog MEI driver uses the kernel watchdog API to configure the Intel AMT Watchdog and to send heartbeats to it. The default timeout of the watchdog is 120 seconds.

If the Intel AMT is not enabled in the firmware then the watchdog client won’t enumerate on the me client bus and watchdog devices won’t be exposed.

—linux-mei@linux.intel.com

CHAPTER

FORTY

MEMORY TECHNOLOGY DEVICE (MTD)

40.1 Upgrading BIOS using intel-spi

Many Intel CPUs like Baytrail and Braswell include SPI serial flash host controller which is used to hold BIOS and other platform specific data. Since contents of the SPI serial flash is crucial for machine to function, it is typically protected by different hardware protection mechanisms to avoid accidental (or on purpose) overwrite of the content.

Not all manufacturers protect the SPI serial flash, mainly because it allows upgrading the BIOS image directly from an OS.

The intel-spi driver makes it possible to read and write the SPI serial flash, if certain protection bits are not set and locked. If it finds any of them set, the whole MTD device is made read-only to prevent partial overwrites. By default the driver exposes SPI serial flash contents as read-only but it can be changed from kernel command line, passing "intel-spi.writeable=1".

Please keep in mind that overwriting the BIOS image on SPI serial flash might render the machine unbootable and requires special equipment like Dediprog to revive. You have been warned!

Below are the steps how to upgrade MinnowBoard MAX BIOS directly from Linux.

1) Download and extract the latest Minnowboard MAX BIOS SPI image [1]. At the time writing this the latest image is v92.
2) Install mtd-utils package [2]. We need this in order to erase the SPI serial flash. Distros like Debian and Fedora have this prepackaged with name "mtd-utils".
3) Add "intel-spi.writeable=1" to the kernel command line and reboot the board (you can also reload the driver passing "writeable=1" as module parameter to modprobe).
4) Once the board is up and running again, find the right MTD partition (it is named as "BIOS"):

```
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00800000 00001000 "BIOS"
```

   So here it will be /dev/mtd0 but it may vary.

5) Make backup of the existing image first:

```
# dd if=/dev/mtd0ro of=bios.bak
16384+0 records in
16384+0 records out
8388608 bytes (8.4 MB) copied, 10.0269 s, 837 kB/s
```

6) Verify the backup:

   # sha1sum /dev/mtd0ro bios.bak fdbb011920572ca6c991377c4b418a0502668b73 /dev/mtd0ro fdbb011920572ca6c991377c4b418a0502668b73 bios.bak

   The SHA1 sums must match. Otherwise do not continue any further!

7) Erase the SPI serial flash. After this step, do not reboot the board! Otherwise it will not start anymore:

```
# flash_erase /dev/mtd0 0 0
Erasing 4 Kibyte @ 7ff000 -- 100 % complete
```

8) Once completed without errors you can write the new BIOS image:

   # dd if=MNW2MAX1.X64.0092.R01.1605221712.bin of=/dev/mtd0

9) Verify that the new content of the SPI serial flash matches the new BIOS image:

```
# sha1sum /dev/mtd0ro MNW2MAX1.X64.0092.R01.1605221712.bin
9b4df9e4be2057fceec3a5529ec3d950836c87a2  /dev/mtd0ro
9b4df9e4be2057fceec3a5529ec3d950836c87a2 MNW2MAX1.X64.0092.R01.
↪1605221712.bin
```

   The SHA1 sums should match.

10) Now you can reboot your board and observe the new BIOS starting up properly.

40.1.1 References

[1] https://firmware.intel.com/sites/default/files/MinnowBoard%2EMAX_%2EX64%2E92%2ER01%2Ezip

[2] http://www.linux-mtd.infradead.org/

40.2 NAND Error-correction Code

40.2.1 Introduction

Having looked at the linux mtd/nand driver and more specific at nand_ecc.c I felt there was room for optimisation. I bashed the code for a few hours performing tricks like table lookup removing superfluous code etc. After that the speed was increased by 35-40%. Still I was not too happy as I felt there was additional room for improvement.

Bad! I was hooked. I decided to annotate my steps in this file. Perhaps it is useful to someone or someone learns something from it.

40.2.2 The problem

NAND flash (at least SLC one) typically has sectors of 256 bytes. However NAND flash is not extremely reliable so some error detection (and sometimes correction) is needed.

This is done by means of a Hamming code. I'll try to explain it in laymans terms (and apologies to all the pro's in the field in case I do not use the right terminology, my coding theory class was almost 30 years ago, and I must admit it was not one of my favourites).

As I said before the ecc calculation is performed on sectors of 256 bytes. This is done by calculating several parity bits over the rows and columns. The parity used is even parity which means that the parity bit = 1 if the data over which the parity is calculated is 1 and the parity bit = 0 if the data over which the parity is calculated is 0. So the total number of bits over the data over which the parity is calculated + the parity bit is even. (see wikipedia if you can't follow this). Parity is often calculated by means of an exclusive or operation, sometimes also referred to as xor. In C the operator for xor is ^

Back to ecc. Let's give a small figure:

byte 0:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp0	rp2	rp4	⋯	rp14
byte 1:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp1	rp2	rp4	⋯	rp14
byte 2:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp0	rp3	rp4	⋯	rp14
byte 3:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp1	rp3	rp4	⋯	rp14
byte 4:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp0	rp2	rp5	⋯	rp14
⋯													
byte 254:	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	rp0	rp3	rp5	⋯	rp15
byte 255:	bit7 cp1 cp3 cp5	bit6 cp0 cp3 cp5	bit5 cp1 cp2 cp5	bit4 cp0 cp2 cp5	bit3 cp1 cp3 cp4	bit2 cp0 cp3 cp4	bit1 cp1 cp2 cp4	bit0 cp0 cp2 cp4	rp1	rp3	rp5	⋯	rp15

This figure represents a sector of 256 bytes. cp is my abbreviation for column parity, rp for row parity.

Let's start to explain column parity.

- cp0 is the parity that belongs to all bit0, bit2, bit4, bit6.

  so the sum of all bit0, bit2, bit4 and bit6 values + cp0 itself is even.

Similarly cp1 is the sum of all bit1, bit3, bit5 and bit7.

- cp2 is the parity over bit0, bit1, bit4 and bit5
- cp3 is the parity over bit2, bit3, bit6 and bit7.
- cp4 is the parity over bit0, bit1, bit2 and bit3.
- cp5 is the parity over bit4, bit5, bit6 and bit7.

Note that each of cp0 .. cp5 is exactly one bit.

Row parity actually works almost the same.

- rp0 is the parity of all even bytes (0, 2, 4, 6, ⋯252, 254)
- rp1 is the parity of all odd bytes (1, 3, 5, 7, ⋯, 253, 255)
- rp2 is the parity of all bytes 0, 1, 4, 5, 8, 9, ⋯(so handle two bytes, then skip 2 bytes).
- rp3 is covers the half rp2 does not cover (bytes 2, 3, 6, 7, 10, 11, ⋯)
- for rp4 the rule is cover 4 bytes, skip 4 bytes, cover 4 bytes, skip 4 etc.

  so rp4 calculates parity over bytes 0, 1, 2, 3, 8, 9, 10, 11, 16, ⋯)
- and rp5 covers the other half, so bytes 4, 5, 6, 7, 12, 13, 14, 15, 20, ..

The story now becomes quite boring. I guess you get the idea.

- rp6 covers 8 bytes then skips 8 etc
- rp7 skips 8 bytes then covers 8 etc
- rp8 covers 16 bytes then skips 16 etc
- rp9 skips 16 bytes then covers 16 etc
- rp10 covers 32 bytes then skips 32 etc
- rp11 skips 32 bytes then covers 32 etc
- rp12 covers 64 bytes then skips 64 etc
- rp13 skips 64 bytes then covers 64 etc
- rp14 covers 128 bytes then skips 128
- rp15 skips 128 bytes then covers 128

In the end the parity bits are grouped together in three bytes as follows:

ECC	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ECC 0	rp07	rp06	rp05	rp04	rp03	rp02	rp01	rp00
ECC 1	rp15	rp14	rp13	rp12	rp11	rp10	rp09	rp08
ECC 2	cp5	cp4	cp3	cp2	cp1	cp0	1	1

I detected after writing this that ST application note AN1823 (http://www.st.com/stonline/) gives a much nicer picture.(but they use line parity as term where I use row parity) Oh well, I’ m graphically challenged, so suffer with me for a moment :-)

And I could not reuse the ST picture anyway for copyright reasons.

40.2.3 Attempt 0

Implementing the parity calculation is pretty simple. In C pseudocode:

```
for (i = 0; i < 256; i++)
{
  if (i & 0x01)
     rp1 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp1;
  else
     rp0 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp0;
  if (i & 0x02)
     rp3 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp3;
  else
     rp2 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp2;
  if (i & 0x04)
    rp5 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp5;
  else
    rp4 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp4;
  if (i & 0x08)
    rp7 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp7;
  else
    rp6 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp6;
  if (i & 0x10)
    rp9 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp9;
  else
    rp8 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp8;
  if (i & 0x20)
    rp11 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp11;
  else
    rp10 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp10;
  if (i & 0x40)
    rp13 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp13;
  else
    rp12 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp12;
  if (i & 0x80)
    rp15 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp15;
  else
    rp14 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ bit3 ^ bit2 ^ bit1 ^ bit0 ^ rp14;
  cp0 = bit6 ^ bit4 ^ bit2 ^ bit0 ^ cp0;
  cp1 = bit7 ^ bit5 ^ bit3 ^ bit1 ^ cp1;
  cp2 = bit5 ^ bit4 ^ bit1 ^ bit0 ^ cp2;
  cp3 = bit7 ^ bit6 ^ bit3 ^ bit2 ^ cp3
  cp4 = bit3 ^ bit2 ^ bit1 ^ bit0 ^ cp4
  cp5 = bit7 ^ bit6 ^ bit5 ^ bit4 ^ cp5
}
```

40.2.4 Analysis 0

C does have bitwise operators but not really operators to do the above efficiently (and most hardware has no such instructions either). Therefore without implementing this it was clear that the code above was not going to bring me a Nobel prize :-)

Fortunately the exclusive or operation is commutative, so we can combine the values in any order. So instead of calculating all the bits individually, let us try to rearrange things. For the column parity this is easy. We can just xor the bytes and in the end filter out the relevant bits. This is pretty nice as it will bring all cp calculation out of the for loop.

Similarly we can first xor the bytes for the various rows. This leads to:

40.2.5 Attempt 1

```
const char parity[256] = {
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1,
    0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0
};

void ecc1(const unsigned char *buf, unsigned char *code)
{
    int i;
    const unsigned char *bp = buf;
    unsigned char cur;
    unsigned char rp0, rp1, rp2, rp3, rp4, rp5, rp6, rp7;
    unsigned char rp8, rp9, rp10, rp11, rp12, rp13, rp14, rp15;
    unsigned char par;

    par = 0;
    rp0 = 0; rp1 = 0; rp2 = 0; rp3 = 0;
    rp4 = 0; rp5 = 0; rp6 = 0; rp7 = 0;
    rp8 = 0; rp9 = 0; rp10 = 0; rp11 = 0;
    rp12 = 0; rp13 = 0; rp14 = 0; rp15 = 0;

    for (i = 0; i < 256; i++)
    {
        cur = *bp++;
        par ^= cur;
```

(continues on next page)

(continued from previous page)

```
        if (i & 0x01) rp1 ^= cur; else rp0 ^= cur;
        if (i & 0x02) rp3 ^= cur; else rp2 ^= cur;
        if (i & 0x04) rp5 ^= cur; else rp4 ^= cur;
        if (i & 0x08) rp7 ^= cur; else rp6 ^= cur;
        if (i & 0x10) rp9 ^= cur; else rp8 ^= cur;
        if (i & 0x20) rp11 ^= cur; else rp10 ^= cur;
        if (i & 0x40) rp13 ^= cur; else rp12 ^= cur;
        if (i & 0x80) rp15 ^= cur; else rp14 ^= cur;
    }
    code[0] =
        (parity[rp7] << 7) |
        (parity[rp6] << 6) |
        (parity[rp5] << 5) |
        (parity[rp4] << 4) |
        (parity[rp3] << 3) |
        (parity[rp2] << 2) |
        (parity[rp1] << 1) |
        (parity[rp0]);
    code[1] =
        (parity[rp15] << 7) |
        (parity[rp14] << 6) |
        (parity[rp13] << 5) |
        (parity[rp12] << 4) |
        (parity[rp11] << 3) |
        (parity[rp10] << 2) |
        (parity[rp9]  << 1) |
        (parity[rp8]);
    code[2] =
        (parity[par & 0xf0] << 7) |
        (parity[par & 0x0f] << 6) |
        (parity[par & 0xcc] << 5) |
        (parity[par & 0x33] << 4) |
        (parity[par & 0xaa] << 3) |
        (parity[par & 0x55] << 2);
    code[0] = ~code[0];
    code[1] = ~code[1];
    code[2] = ~code[2];
}
```

Still pretty straightforward. The last three invert statements are there to give a checksum of 0xff 0xff 0xff for an empty flash. In an empty flash all data is 0xff, so the checksum then matches.

I also introduced the parity lookup. I expected this to be the fastest way to calculate the parity, but I will investigate alternatives later on.

40.2.6 Analysis 1

The code works, but is not terribly efficient. On my system it took almost 4 times as much time as the linux driver code. But hey, if it was that easy this would have been done long before. No pain. no gain.

Fortunately there is plenty of room for improvement.

In step 1 we moved from bit-wise calculation to byte-wise calculation. However in C we can also use the unsigned long data type and virtually every modern microprocessor supports 32 bit operations, so why not try to write our code in such a way that we process data in 32 bit chunks.

Of course this means some modification as the row parity is byte by byte. A quick analysis: for the column parity we use the par variable. When extending to 32 bits we can in the end easily calculate rp0 and rp1 from it. (because par now consists of 4 bytes, contributing to rp1, rp0, rp1, rp0 respectively, from MSB to LSB) also rp2 and rp3 can be easily retrieved from par as rp3 covers the first two MSBs and rp2 covers the last two LSBs.

Note that of course now the loop is executed only 64 times (256/4). And note that care must taken wrt byte ordering. The way bytes are ordered in a long is machine dependent, and might affect us. Anyway, if there is an issue: this code is developed on x86 (to be precise: a DELL PC with a D920 Intel CPU)

And of course the performance might depend on alignment, but I expect that the I/O buffers in the nand driver are aligned properly (and otherwise that should be fixed to get maximum performance).

Let’s give it a try…

40.2.7 Attempt 2

```
extern const char parity[256];

void ecc2(const unsigned char *buf, unsigned char *code)
{
    int i;
    const unsigned long *bp = (unsigned long *)buf;
    unsigned long cur;
    unsigned long rp0, rp1, rp2, rp3, rp4, rp5, rp6, rp7;
    unsigned long rp8, rp9, rp10, rp11, rp12, rp13, rp14, rp15;
    unsigned long par;

    par = 0;
    rp0 = 0; rp1 = 0; rp2 = 0; rp3 = 0;
    rp4 = 0; rp5 = 0; rp6 = 0; rp7 = 0;
    rp8 = 0; rp9 = 0; rp10 = 0; rp11 = 0;
    rp12 = 0; rp13 = 0; rp14 = 0; rp15 = 0;

    for (i = 0; i < 64; i++)
    {
        cur = *bp++;
        par ^= cur;
        if (i & 0x01) rp5 ^= cur; else rp4 ^= cur;
```

(continues on next page)

(continued from previous page)

```
        if (i & 0x02) rp7 ^= cur; else rp6 ^= cur;
        if (i & 0x04) rp9 ^= cur; else rp8 ^= cur;
        if (i & 0x08) rp11 ^= cur; else rp10 ^= cur;
        if (i & 0x10) rp13 ^= cur; else rp12 ^= cur;
        if (i & 0x20) rp15 ^= cur; else rp14 ^= cur;
    }
    /*
       we need to adapt the code generation for the fact that rp vars are␣
↪now
       long; also the column parity calculation needs to be changed.
       we'll bring rp4 to 15 back to single byte entities by shifting and
       xoring
    */
    rp4 ^= (rp4 >> 16); rp4 ^= (rp4 >> 8); rp4 &= 0xff;
    rp5 ^= (rp5 >> 16); rp5 ^= (rp5 >> 8); rp5 &= 0xff;
    rp6 ^= (rp6 >> 16); rp6 ^= (rp6 >> 8); rp6 &= 0xff;
    rp7 ^= (rp7 >> 16); rp7 ^= (rp7 >> 8); rp7 &= 0xff;
    rp8 ^= (rp8 >> 16); rp8 ^= (rp8 >> 8); rp8 &= 0xff;
    rp9 ^= (rp9 >> 16); rp9 ^= (rp9 >> 8); rp9 &= 0xff;
    rp10 ^= (rp10 >> 16); rp10 ^= (rp10 >> 8); rp10 &= 0xff;
    rp11 ^= (rp11 >> 16); rp11 ^= (rp11 >> 8); rp11 &= 0xff;
    rp12 ^= (rp12 >> 16); rp12 ^= (rp12 >> 8); rp12 &= 0xff;
    rp13 ^= (rp13 >> 16); rp13 ^= (rp13 >> 8); rp13 &= 0xff;
    rp14 ^= (rp14 >> 16); rp14 ^= (rp14 >> 8); rp14 &= 0xff;
    rp15 ^= (rp15 >> 16); rp15 ^= (rp15 >> 8); rp15 &= 0xff;
    rp3 = (par >> 16); rp3 ^= (rp3 >> 8); rp3 &= 0xff;
    rp2 = par & 0xffff; rp2 ^= (rp2 >> 8); rp2 &= 0xff;
    par ^= (par >> 16);
    rp1 = (par >> 8); rp1 &= 0xff;
    rp0 = (par & 0xff);
    par ^= (par >> 8); par &= 0xff;

    code[0] =
        (parity[rp7] << 7) |
        (parity[rp6] << 6) |
        (parity[rp5] << 5) |
        (parity[rp4] << 4) |
        (parity[rp3] << 3) |
        (parity[rp2] << 2) |
        (parity[rp1] << 1) |
        (parity[rp0]);
    code[1] =
        (parity[rp15] << 7) |
        (parity[rp14] << 6) |
        (parity[rp13] << 5) |
        (parity[rp12] << 4) |
        (parity[rp11] << 3) |
        (parity[rp10] << 2) |
        (parity[rp9]  << 1) |
        (parity[rp8]);
    code[2] =
        (parity[par & 0xf0] << 7) |
        (parity[par & 0x0f] << 6) |
        (parity[par & 0xcc] << 5) |
        (parity[par & 0x33] << 4) |
        (parity[par & 0xaa] << 3) |
```

(continues on next page)

(continued from previous page)

```
        (parity[par & 0x55] << 2);
    code[0] = ~code[0];
    code[1] = ~code[1];
    code[2] = ~code[2];
}
```

The parity array is not shown any more. Note also that for these examples I kinda deviated from my regular programming style by allowing multiple statements on a line, not using { } in then and else blocks with only a single statement and by using operators like ^=

40.2.8 Analysis 2

The code (of course) works, and hurray: we are a little bit faster than the linux driver code (about 15%). But wait, don't cheer too quickly. There is more to be gained. If we look at e.g. rp14 and rp15 we see that we either xor our data with rp14 or with rp15. However we also have par which goes over all data. This means there is no need to calculate rp14 as it can be calculated from rp15 through rp14 = par ^ rp15, because par = rp14 ^ rp15; (or if desired we can avoid calculating rp15 and calculate it from rp14). That is why some places refer to inverse parity. Of course the same thing holds for rp4/5, rp6/7, rp8/9, rp10/11 and rp12/13. Effectively this means we can eliminate the else clause from the if statements. Also we can optimise the calculation in the end a little bit by going from long to byte first. Actually we can even avoid the table lookups

40.2.9 Attempt 3

Odd replaced:

```
if (i & 0x01) rp5 ^= cur; else rp4 ^= cur;
if (i & 0x02) rp7 ^= cur; else rp6 ^= cur;
if (i & 0x04) rp9 ^= cur; else rp8 ^= cur;
if (i & 0x08) rp11 ^= cur; else rp10 ^= cur;
if (i & 0x10) rp13 ^= cur; else rp12 ^= cur;
if (i & 0x20) rp15 ^= cur; else rp14 ^= cur;
```

with:

```
if (i & 0x01) rp5 ^= cur;
if (i & 0x02) rp7 ^= cur;
if (i & 0x04) rp9 ^= cur;
if (i & 0x08) rp11 ^= cur;
if (i & 0x10) rp13 ^= cur;
if (i & 0x20) rp15 ^= cur;
```

and outside the loop added:

```
rp4  = par ^ rp5;
rp6  = par ^ rp7;
rp8  = par ^ rp9;
rp10  = par ^ rp11;
```

(continues on next page)

(continued from previous page)

```
rp12  = par ^ rp13;
rp14  = par ^ rp15;
```

And after that the code takes about 30% more time, although the number of statements is reduced. This is also reflected in the assembly code.

40.2.10 Analysis 3

Very weird. Guess it has to do with caching or instruction parallellism or so. I also tried on an eeePC (Celeron, clocked at 900 Mhz). Interesting observation was that this one is only 30% slower (according to time) executing the code as my 3Ghz D920 processor.

Well, it was expected not to be easy so maybe instead move to a different track: let's move back to the code from attempt2 and do some loop unrolling. This will eliminate a few if statements. I'll try different amounts of unrolling to see what works best.

40.2.11 Attempt 4

Unrolled the loop 1, 2, 3 and 4 times. For 4 the code starts with:

```
for (i = 0; i < 4; i++)
{
    cur = *bp++;
    par ^= cur;
    rp4 ^= cur;
    rp6 ^= cur;
    rp8 ^= cur;
    rp10 ^= cur;
    if (i & 0x1) rp13 ^= cur; else rp12 ^= cur;
    if (i & 0x2) rp15 ^= cur; else rp14 ^= cur;
    cur = *bp++;
    par ^= cur;
    rp5 ^= cur;
    rp6 ^= cur;
    ...
```

40.2.12 Analysis 4

Unrolling once gains about 15%

Unrolling twice keeps the gain at about 15%

Unrolling three times gives a gain of 30% compared to attempt 2.

Unrolling four times gives a marginal improvement compared to unrolling three times.

I decided to proceed with a four time unrolled loop anyway. It was my gut feeling that in the next steps I would obtain additional gain from it.

The next step was triggered by the fact that par contains the xor of all bytes and rp4 and rp5 each contain the xor of half of the bytes. So in effect par = rp4 ^ rp5. But as xor is commutative we can also say that rp5 = par ^ rp4. So no need to keep both rp4 and rp5 around. We can eliminate rp5 (or rp4, but I already foresaw another optimisation). The same holds for rp6/7, rp8/9, rp10/11 rp12/13 and rp14/15.

40.2.13 Attempt 5

Effectively so all odd digit rp assignments in the loop were removed. This included the else clause of the if statements. Of course after the loop we need to correct things by adding code like:

```
rp5 = par ^ rp4;
```

Also the initial assignments (rp5 = 0; etc) could be removed. Along the line I also removed the initialisation of rp0/1/2/3.

40.2.14 Analysis 5

Measurements showed this was a good move. The run-time roughly halved compared with attempt 4 with 4 times unrolled, and we only require 1/3rd of the processor time compared to the current code in the linux kernel.

However, still I thought there was more. I didn't like all the if statements. Why not keep a running parity and only keep the last if statement. Time for yet another version!

40.2.15 Attempt 6

THe code within the for loop was changed to:

```
for (i = 0; i < 4; i++)
{
    cur = *bp++; tmppar  = cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= tmppar;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp8 ^= tmppar;

    cur = *bp++; tmppar ^= cur; rp4 ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp10 ^= tmppar;

    cur = *bp++; tmppar ^= cur; rp4 ^= cur; rp6 ^= cur; rp8 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur; rp8 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur; rp8 ^= cur;
    cur = *bp++; tmppar ^= cur; rp8 ^= cur;

    cur = *bp++; tmppar ^= cur; rp4 ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
```

(continues on next page)

(continued from previous page)

```
    cur = *bp++; tmppar ^= cur;

    par ^= tmppar;
    if ((i & 0x1) == 0) rp12 ^= tmppar;
    if ((i & 0x2) == 0) rp14 ^= tmppar;
}
```

As you can see tmppar is used to accumulate the parity within a for iteration. In the last 3 statements is added to par and, if needed, to rp12 and rp14.

While making the changes I also found that I could exploit that tmppar contains the running parity for this iteration. So instead of having: rp4 ^= cur; rp6 ^= cur; I removed the rp6 ^= cur; statement and did rp6 ^= tmppar; on next statement. A similar change was done for rp8 and rp10

40.2.16 Analysis 6

Measuring this code again showed big gain. When executing the original linux code 1 million times, this took about 1 second on my system. (using time to measure the performance). After this iteration I was back to 0.075 sec. Actually I had to decide to start measuring over 10 million iterations in order not to lose too much accuracy. This one definitely seemed to be the jackpot!

There is a little bit more room for improvement though. There are three places with statements:

```
rp4 ^= cur; rp6 ^= cur;
```

It seems more efficient to also maintain a variable rp4_6 in the while loop; This eliminates 3 statements per loop. Of course after the loop we need to correct by adding:

```
rp4 ^= rp4_6;
rp6 ^= rp4_6
```

Furthermore there are 4 sequential assignments to rp8. This can be encoded slightly more efficiently by saving tmppar before those 4 lines and later do rp8 = rp8 ^ tmppar ^ notrp8; (where notrp8 is the value of rp8 before those 4 lines). Again a use of the commutative property of xor. Time for a new test!

40.2.17 Attempt 7

The new code now looks like:

```
for (i = 0; i < 4; i++)
{
    cur = *bp++; tmppar  = cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= tmppar;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp8 ^= tmppar;
```

(continues on next page)

(continued from previous page)

```
    cur = *bp++; tmppar ^= cur; rp4_6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur; rp10 ^= tmppar;

    notrp8 = tmppar;
    cur = *bp++; tmppar ^= cur; rp4_6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur;
    rp8 = rp8 ^ tmppar ^ notrp8;

    cur = *bp++; tmppar ^= cur; rp4_6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp6 ^= cur;
    cur = *bp++; tmppar ^= cur; rp4 ^= cur;
    cur = *bp++; tmppar ^= cur;

    par ^= tmppar;
    if ((i & 0x1) == 0) rp12 ^= tmppar;
    if ((i & 0x2) == 0) rp14 ^= tmppar;
}
rp4 ^= rp4_6;
rp6 ^= rp4_6;
```

Not a big change, but every penny counts :-)

40.2.18 Analysis 7

Actually this made things worse. Not very much, but I don't want to move into the wrong direction. Maybe something to investigate later. Could have to do with caching again.

Guess that is what there is to win within the loop. Maybe unrolling one more time will help. I'll keep the optimisations from 7 for now.

40.2.19 Attempt 8

Unrolled the loop one more time.

40.2.20 Analysis 8

This makes things worse. Let's stick with attempt 6 and continue from there. Although it seems that the code within the loop cannot be optimised further there is still room to optimize the generation of the ecc codes. We can simply calculate the total parity. If this is 0 then rp4 = rp5 etc. If the parity is 1, then rp4 = !rp5;

But if rp4 = rp5 we do not need rp5 etc. We can just write the even bits in the result byte and then do something like:

```
code[0] |= (code[0] << 1);
```

Lets test this.

40.2.21 Attempt 9

Changed the code but again this slightly degrades performance. Tried all kind of other things, like having dedicated parity arrays to avoid the shift after parity[rp7] << 7; No gain. Change the lookup using the parity array by using shift operators (e.g. replace parity[rp7] << 7 with:

```
rp7 ^= (rp7 << 4);
rp7 ^= (rp7 << 2);
rp7 ^= (rp7 << 1);
rp7 &= 0x80;
```

No gain.

The only marginal change was inverting the parity bits, so we can remove the last three invert statements.

Ah well, pity this does not deliver more. Then again 10 million iterations using the linux driver code takes between 13 and 13.5 seconds, whereas my code now takes about 0.73 seconds for those 10 million iterations. So basically I've improved the performance by a factor 18 on my system. Not that bad. Of course on different hardware you will get different results. No warranties!

But of course there is no such thing as a free lunch. The codesize almost tripled (from 562 bytes to 1434 bytes). Then again, it is not that much.

40.2.22 Correcting errors

For correcting errors I again used the ST application note as a starter, but I also peeked at the existing code.

The algorithm itself is pretty straightforward. Just xor the given and the calculated ecc. If all bytes are 0 there is no problem. If 11 bits are 1 we have one correctable bit error. If there is 1 bit 1, we have an error in the given ecc code.

It proved to be fastest to do some table lookups. Performance gain introduced by this is about a factor 2 on my system when a repair had to be done, and 1% or so if no repair had to be done.

Code size increased from 330 bytes to 686 bytes for this function. (gcc 4.2, -O3)

40.2.23 Conclusion

The gain when calculating the ecc is tremendous. Om my development hardware a speedup of a factor of 18 for ecc calculation was achieved. On a test on an embedded system with a MIPS core a factor 7 was obtained.

On a test with a Linksys NSLU2 (ARMv5TE processor) the speedup was a factor 5 (big endian mode, gcc 4.1.2, -O3)

For correction not much gain could be obtained (as bitflips are rare). Then again there are also much less cycles spent there.

It seems there is not much more gain possible in this, at least when programmed in C. Of course it might be possible to squeeze something more out of it with an

assembler program, but due to pipeline behaviour etc this is very tricky (at least for intel hw).

Author: Frans Meulenbroeks

40.3 SPI NOR framework

40.3.1 Part I - Why do we need this framework?

SPI bus controllers (drivers/spi/) only deal with streams of bytes; the bus controller operates agnostic of the specific device attached. However, some controllers (such as Freescale's QuadSPI controller) cannot easily handle arbitrary streams of bytes, but rather are designed specifically for SPI NOR.

In particular, Freescale' s QuadSPI controller must know the NOR commands to find the right LUT sequence. Unfortunately, the SPI subsystem has no notion of opcodes, addresses, or data payloads; a SPI controller simply knows to send or receive bytes (Tx and Rx). Therefore, we must define a new layering scheme under which the controller driver is aware of the opcodes, addressing, and other details of the SPI NOR protocol.

40.3.2 Part II - How does the framework work?

This framework just adds a new layer between the MTD and the SPI bus driver. With this new layer, the SPI NOR controller driver does not depend on the m25p80 code anymore.

Before this framework, the layer is like:

```
                MTD
       ------------------------
                m25p80
       ------------------------
              SPI bus driver
       ------------------------
               SPI NOR chip

 After this framework, the layer is like:
                MTD
       ------------------------
             SPI NOR framework
       ------------------------
                m25p80
       ------------------------
              SPI bus driver
       ------------------------
              SPI NOR chip

With the SPI NOR controller driver (Freescale QuadSPI), it looks like:
                MTD
       ------------------------
```

(continues on next page)

(continued from previous page)

```
            SPI NOR framework
      ------------------------
             fsl-quadSPI
      ------------------------
            SPI NOR chip
```

40.3.3 Part III - How can drivers use the framework?

The main API is spi_nor_scan(). Before you call the hook, a driver should initialize the necessary fields for spi_nor{}. Please see drivers/mtd/spi-nor/spi-nor.c for detail. Please also refer to spi-fsl-qspi.c when you want to write a new driver for a SPI NOR controller. Another API is spi_nor_restore(), this is used to restore the status of SPI flash chip such as addressing mode. Call it whenever detach the driver from device or reboot the system.

CHAPTER

FORTYONE

MMC/SD/SDIO CARD SUPPORT

41.1 SD and MMC Block Device Attributes

These attributes are defined for the block devices associated with the SD or MMC device.

The following attributes are read/write.

force_ro	Enforce read-only access even if write protect switch is off.

41.1.1 SD and MMC Device Attributes

All attributes are read-only.

cid	Card Identification Register
csd	Card Specific Data Register
scr	SD Card Configuration Register (SD only)
date	Manufacturing Date (from CID Register)
fwrev	Firmware/Product Revision (from CID Register) (SD and MMCv1 only)
hwrev	Hardware/Product Revision (from CID Register) (SD and MMCv1 only)
manfid	Manufacturer ID (from CID Register)
name	Product Name (from CID Register)
oemid	OEM/Application ID (from CID Register)
prv	Product Revision (from CID Register) (SD and MMCv4 only)
serial	Product Serial Number (from CID Register)
erase_size	Erase group size
preferred_erase_size	Preferred erase size
raw_rpmb_size_mult	RPMB partition size
rel_sectors	Reliable write sector count
ocr	Operation Conditions Register
dsr	Driver Stage Register
cmdq_en	Command Queue enabled: 1 => enabled, 0 => not enabled

Note on Erase Size and Preferred Erase Size:

> “erase_size” is the minimum size, in bytes, of an erase operation. For MMC, “erase_size” is the erase group size reported by the card. Note that “erase_size” does not apply to trim or secure trim operations where the minimum size is always one 512 byte sector. For SD, “erase_size” is 512 if the card is block-addressed, 0 otherwise.
>
> SD/MMC cards can erase an arbitrarily large area up to and including the whole card. When erasing a large area it may be desirable to do it in smaller chunks for three reasons:
>
> 1. A single erase command will make all other I/O on the card wait. This is not a problem if the whole card is being erased, but erasing one partition will make I/O for another partition on the same card wait for the duration of the erase - which could be a several minutes.
> 2. To be able to inform the user of erase progress.

3. The erase timeout becomes too large to be very useful. Because the erase timeout contains a margin which is multiplied by the size of the erase area, the value can end up being several minutes for large areas.

"erase_size"is not the most efficient unit to erase (especially for SD where it is just one sector), hence "preferred_erase_size"provides a good chunk size for erasing large areas.

For MMC, "preferred_erase_size"is the high-capacity erase size if a card specifies one, otherwise it is based on the capacity of the card.

For SD, "preferred_erase_size" is the allocation unit size specified by the card.

"preferred_erase_size" is in bytes.

Note on raw_rpmb_size_mult:

"raw_rpmb_size_mult" is a multiple of 128kB block.

RPMB size in byte is calculated by using the following equation:

RPMB partition size = 128kB x raw_rpmb_size_mult

41.2 SD and MMC Device Partitions

Device partitions are additional logical block devices present on the SD/MMC device.

As of this writing, MMC boot partitions as supported and exposed as /dev/mmcblkXboot0 and /dev/mmcblkXboot1, where X is the index of the parent /dev/mmcblkX.

41.2.1 MMC Boot Partitions

Read and write access is provided to the two MMC boot partitions. Due to the sensitive nature of the boot partition contents, which often store a bootloader or bootloader configuration tables crucial to booting the platform, write access is disabled by default to reduce the chance of accidental bricking.

To enable write access to /dev/mmcblkXbootY, disable the forced read-only access with:

```
echo 0 > /sys/block/mmcblkXbootY/force_ro
```

To re-enable read-only access:

```
echo 1 > /sys/block/mmcblkXbootY/force_ro
```

The boot partitions can also be locked read only until the next power on, with:

```
echo 1 > /sys/block/mmcblkXbootY/ro_lock_until_next_power_on
```

This is a feature of the card and not of the kernel. If the card does not support boot partition locking, the file will not exist. If the feature has been disabled on the card, the file will be read-only.

The boot partitions can also be locked permanently, but this feature is not accessible through sysfs in order to avoid accidental or malicious bricking.

41.3 MMC Asynchronous Request

41.3.1 Rationale

How significant is the cache maintenance overhead?

It depends. Fast eMMC and multiple cache levels with speculative cache pre-fetch makes the cache overhead relatively significant. If the DMA preparations for the next request are done in parallel with the current transfer, the DMA preparation overhead would not affect the MMC performance.

The intention of non-blocking (asynchronous) MMC requests is to minimize the time between when an MMC request ends and another MMC request begins.

Using mmc_wait_for_req(), the MMC controller is idle while dma_map_sg and dma_unmap_sg are processing. Using non-blocking MMC requests makes it possible to prepare the caches for next job in parallel with an active MMC request.

41.3.2 MMC block driver

The mmc_blk_issue_rw_rq() in the MMC block driver is made non-blocking.

The increase in throughput is proportional to the time it takes to prepare (major part of preparations are dma_map_sg() and dma_unmap_sg()) a request and how fast the memory is. The faster the MMC/SD is the more significant the prepare request time becomes. Roughly the expected performance gain is 5% for large writes and 10% on large reads on a L2 cache platform. In power save mode, when clocks run on a lower frequency, the DMA preparation may cost even more. As long as these slower preparations are run in parallel with the transfer performance won' t be affected.

41.3.3 Details on measurements from IOZone and mmc_test

https://wiki.linaro.org/WorkingGroups/Kernel/Specs/StoragePerfMMC-async-req

41.3.4 MMC core API extension

There is one new public function mmc_start_req().

It starts a new MMC command request for a host. The function isn't truly non-blocking. If there is an ongoing async request it waits for completion of that request and starts the new one and returns. It doesn't wait for the new request to complete. If there is no ongoing request it starts the new request and returns immediately.

41.3.5 MMC host extensions

There are two optional members in the mmc_host_ops – pre_req() and post_req() – that the host driver may implement in order to move work to before and after the actual mmc_host_ops.request() function is called.

In the DMA case pre_req() may do dma_map_sg() and prepare the DMA descriptor, and post_req() runs the dma_unmap_sg().

41.3.6 Optimize for the first request

The first request in a series of requests can't be prepared in parallel with the previous transfer, since there is no previous request.

The argument is_first_req in pre_req() indicates that there is no previous request. The host driver may optimize for this scenario to minimize the performance loss. A way to optimize for this is to split the current request in two chunks, prepare the first chunk and start the request, and finally prepare the second chunk and start the transfer.

Pseudocode to handle is_first_req scenario with minimal prepare overhead:

```
if (is_first_req && req->size > threshold)
   /* start MMC transfer for the complete transfer size */
   mmc_start_command(MMC_CMD_TRANSFER_FULL_SIZE);

   /*
    * Begin to prepare DMA while cmd is being processed by MMC.
    * The first chunk of the request should take the same time
    * to prepare as the "MMC process command time".
    * If prepare time exceeds MMC cmd time
    * the transfer is delayed, guesstimate max 4k as first chunk size.
    */
    prepare_1st_chunk_for_dma(req);
    /* flush pending desc to the DMAC (dmaengine.h) */
    dma_issue_pending(req->dma_desc);

    prepare_2nd_chunk_for_dma(req);
    /*
     * The second issue_pending should be called before MMC runs out
     * of the first chunk. If the MMC runs out of the first data chunk
     * before this call, the transfer is delayed.
     */
    dma_issue_pending(req->dma_desc);
```

41.4 MMC tools introduction

There is one MMC test tools called mmc-utils, which is maintained by Chris Ball, you can find it at the below public git repository:

http://git.kernel.org/cgit/linux/kernel/git/cjb/mmc-utils.git/

41.4.1 Functions

The mmc-utils tools can do the following:

- Print and parse extcsd data.
- Determine the eMMC writeprotect status.
- Set the eMMC writeprotect status.
- Set the eMMC data sector size to 4KB by disabling emulation.
- Create general purpose partition.
- Enable the enhanced user area.
- Enable write reliability per partition.
- Print the response to STATUS_SEND (CMD13).
- Enable the boot partition.
- Set Boot Bus Conditions.
- Enable the eMMC BKOPS feature.
- Permanently enable the eMMC H/W Reset feature.
- Permanently disable the eMMC H/W Reset feature.
- Send Sanitize command.
- Program authentication key for the device.
- Counter value for the rpmb device will be read to stdout.
- Read from rpmb device to output.
- Write to rpmb device from data file.
- Enable the eMMC cache feature.
- Disable the eMMC cache feature.
- Print and parse CID data.
- Print and parse CSD data.
- Print and parse SCR data.

CHAPTER

FORTYTWO

NON-VOLATILE MEMORY DEVICE (NVDIMM)

42.1 LIBNVDIMM: Non-Volatile Devices

libnvdimm - kernel / libndctl - userspace helper library

linux-nvdimm@lists.01.org

Version 13

42.1.1 Glossary

PMEM: A system-physical-address range where writes are persistent. A block device composed of PMEM is capable of DAX. A PMEM address range may span an interleave of several DIMMs.

BLK: A set of one or more programmable memory mapped apertures provided by a DIMM to access its media. This indirection precludes the performance benefit of interleaving, but enables DIMM-bounded failure modes.

DPA: DIMM Physical Address, is a DIMM-relative offset. With one DIMM in the system there would be a 1:1 system-physical-address:DPA association. Once more DIMMs are added a memory controller interleave must be decoded to determine the DPA associated with a given system-physical-address. BLK capacity always has a 1:1 relationship with a single-DIMM's DPA range.

DAX: File system extensions to bypass the page cache and block layer to mmap persistent memory, from a PMEM block device, directly into a process address space.

DSM: Device Specific Method: ACPI method to to control specific device - in this case the firmware.

DCR: NVDIMM Control Region Structure defined in ACPI 6 Section 5.2.25.5. It defines a vendor-id, device-id, and interface format for a given DIMM.

BTT: Block Translation Table: Persistent memory is byte addressable. Existing software may have an expectation that the power-fail-atomicity of writes is at least one sector, 512 bytes. The BTT is an indirection table with atomic update semantics to front a PMEM/BLK block device driver and present arbitrary atomic sector sizes.

LABEL: Metadata stored on a DIMM device that partitions and identifies (persistently names) storage between PMEM and BLK. It also partitions BLK stor-

age to host BTTs with different parameters per BLK-partition. Note that traditional partition tables, GPT/MBR, are layered on top of a BLK or PMEM device.

42.1.2 Overview

The LIBNVDIMM subsystem provides support for three types of NVDIMMs, namely, PMEM, BLK, and NVDIMM devices that can simultaneously support both PMEM and BLK mode access. These three modes of operation are described by the "NVDIMM Firmware Interface Table"(NFIT) in ACPI 6. While the LIBNVDIMM implementation is generic and supports pre-NFIT platforms, it was guided by the superset of capabilities need to support this ACPI 6 definition for NVDIMM resources. The bulk of the kernel implementation is in place to handle the case where DPA accessible via PMEM is aliased with DPA accessible via BLK. When that occurs a LABEL is needed to reserve DPA for exclusive access via one mode a time.

Supporting Documents

ACPI 6: http://www.uefi.org/sites/default/files/resources/ACPI_6.0.pdf

NVDIMM Namespace: http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf

DSM Interface Example: http://pmem.io/documents/NVDIMM_DSM_Interface_Example.pdf

Driver Writer's Guide: http://pmem.io/documents/NVDIMM_Driver_Writers_Guide.pdf

Git Trees

LIBNVDIMM: https://git.kernel.org/cgit/linux/kernel/git/djbw/nvdimm.git

LIBNDCTL: https://github.com/pmem/ndctl.git

PMEM: https://github.com/01org/prd

42.1.3 LIBNVDIMM PMEM and BLK

Prior to the arrival of the NFIT, non-volatile memory was described to a system in various ad-hoc ways. Usually only the bare minimum was provided, namely, a single system-physical-address range where writes are expected to be durable after a system power loss. Now, the NFIT specification standardizes not only the description of PMEM, but also BLK and platform message-passing entry points for control and configuration.

For each NVDIMM access method (PMEM, BLK), LIBNVDIMM provides a block device driver:

1. PMEM (nd_pmem.ko): Drives a system-physical-address range. This range is contiguous in system memory and may be interleaved (hardware memory

controller striped) across multiple DIMMs. When interleaved the platform may optionally provide details of which DIMMs are participating in the interleave.

Note that while LIBNVDIMM describes system-physical-address ranges that may alias with BLK access as ND_NAMESPACE_PMEM ranges and those without alias as ND_NAMESPACE_IO ranges, to the nd_pmem driver there is no distinction. The different device-types are an implementation detail that userspace can exploit to implement policies like "only interface with address ranges from certain DIMMs". It is worth noting that when aliasing is present and a DIMM lacks a label, then no block device can be created by default as userspace needs to do at least one allocation of DPA to the PMEM range. In contrast ND_NAMESPACE_IO ranges, once registered, can be immediately attached to nd_pmem.

2. BLK (nd_blk.ko): This driver performs I/O using a set of platform defined apertures. A set of apertures will access just one DIMM. Multiple windows (apertures) allow multiple concurrent accesses, much like tagged-command-queuing, and would likely be used by different threads or different CPUs.

   The NFIT specification defines a standard format for a BLK-aperture, but the spec also allows for vendor specific layouts, and non-NFIT BLK implementations may have other designs for BLK I/O. For this reason "nd_blk" calls back into platform-specific code to perform the I/O.

   One such implementation is defined in the "Driver Writer's Guide" and "DSM Interface Example".

42.1.4 Why BLK?

While PMEM provides direct byte-addressable CPU-load/store access to NVDIMM storage, it does not provide the best system RAS (recovery, availability, and serviceability) model. An access to a corrupted system-physical-address address causes a CPU exception while an access to a corrupted address through an BLK-aperture causes that block window to raise an error status in a register. The latter is more aligned with the standard error model that host-bus-adapter attached disks present.

Also, if an administrator ever wants to replace a memory it is easier to service a system at DIMM module boundaries. Compare this to PMEM where data could be interleaved in an opaque hardware specific manner across several DIMMs.

PMEM vs BLK

BLK-apertures solve these RAS problems, but their presence is also the major contributing factor to the complexity of the ND subsystem. They complicate the implementation because PMEM and BLK alias in DPA space. Any given DIMM's DPA-range may contribute to one or more system-physical-address sets of interleaved DIMMs, and may also be accessed in its entirety through its BLK-aperture. Accessing a DPA through a system-physical-address while simultaneously accessing the same DPA through a BLK-aperture has undefined results. For this reason, DIMMs with this dual interface configuration include a DSM function to store/retrieve

a LABEL. The LABEL effectively partitions the DPA-space into exclusive system-physical-address and BLK-aperture accessible regions. For simplicity a DIMM is allowed a PMEM "region" per each interleave set in which it is a member. The remaining DPA space can be carved into an arbitrary number of BLK devices with discontiguous extents.

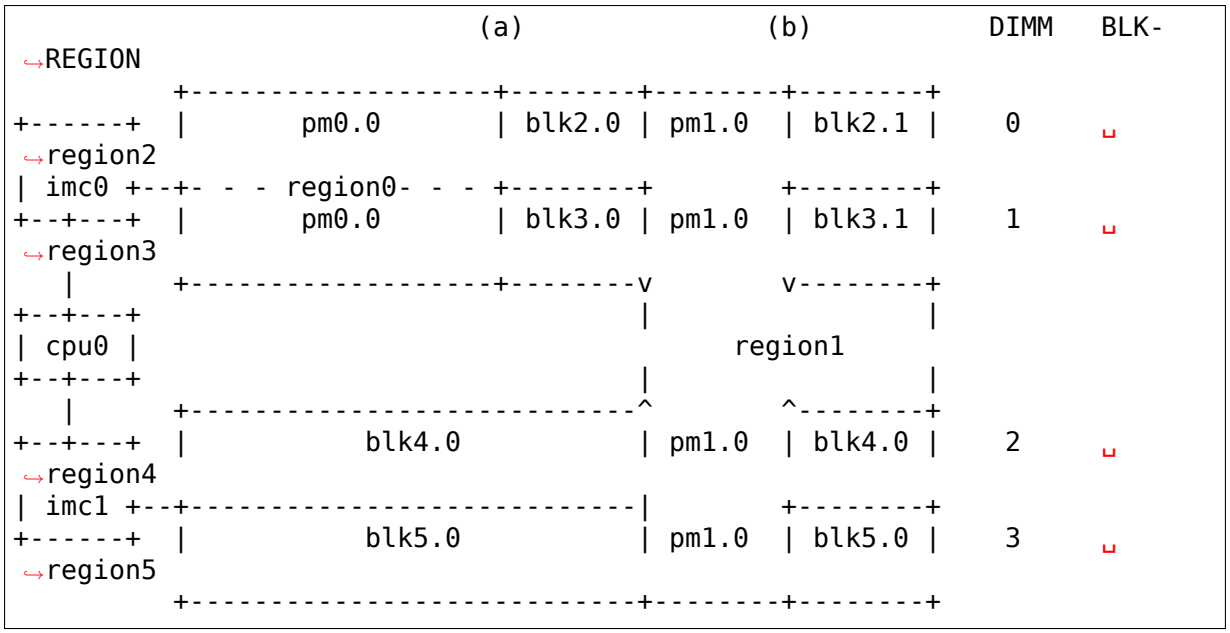
BLK-REGIONs, PMEM-REGIONs, Atomic Sectors, and DAX

One of the few reasons to allow multiple BLK namespaces per REGION is so that each BLK-namespace can be configured with a BTT with unique atomic sector sizes. While a PMEM device can host a BTT the LABEL specification does not provide for a sector size to be specified for a PMEM namespace.

This is due to the expectation that the primary usage model for PMEM is via DAX, and the BTT is incompatible with DAX. However, for the cases where an application or filesystem still needs atomic sector update guarantees it can register a BTT on a PMEM device or partition. See LIBNVDIMM/NDCTL: Block Translation Table "btt"

42.1.5 Example NVDIMM Platform

For the remainder of this document the following diagram will be referenced for any example sysfs layouts:



In this platform we have four DIMMs and two memory controllers in one socket. Each unique interface (BLK or PMEM) to DPA space is identified by a region device with a dynamically assigned id (REGION0 - REGION5).

1. The first portion of DIMM0 and DIMM1 are interleaved as REGION0. A single PMEM namespace is created in the REGION0-SPA-range that spans most of DIMM0 and DIMM1 with a user-specified name of "pm0.0". Some of that interleaved system-physical-address

range is reclaimed as BLK-aperture accessed space starting at DPA-offset (a) into each DIMM. In that reclaimed space we create two BLK-aperture "namespaces" from REGION2 and REGION3 where "blk2.0" and "blk3.0" are just human readable names that could be set to any user-desired name in the LABEL.

2. In the last portion of DIMM0 and DIMM1 we have an interleaved system-physical-address range, REGION1, that spans those two DIMMs as well as DIMM2 and DIMM3. Some of REGION1 is allocated to a PMEM namespace named "pm1.0", the rest is reclaimed in 4 BLK-aperture namespaces (for each DIMM in the interleave set), "blk2.1", "blk3.1", "blk4.0", and "blk5.0".

3. The portion of DIMM2 and DIMM3 that do not participate in the REGION1 interleaved system-physical-address range (i.e. the DPA address past offset (b) are also included in the "blk4.0" and "blk5.0" namespaces. Note, that this example shows that BLK-aperture namespaces don't need to be contiguous in DPA-space.

This bus is provided by the kernel under the device /sys/devices/platform/nfit_test.0 when the nfit_test.ko module from tools/testing/nvdimm is loaded. This not only test LIBNVDIMM but the acpi_nfit.ko driver as well.

42.1.6 LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API

What follows is a description of the LIBNVDIMM sysfs layout and a corresponding object hierarchy diagram as viewed through the LIBNDCTL API. The example sysfs paths and diagrams are relative to the Example NVDIMM Platform which is also the LIBNVDIMM bus used in the LIBNDCTL unit test.

LIBNDCTL: Context

Every API call in the LIBNDCTL library requires a context that holds the logging parameters and other library instance state. The library is based on the libabc template:

https://git.kernel.org/cgit/linux/kernel/git/kay/libabc.git

LIBNDCTL: instantiate a new library context example

```
struct ndctl_ctx *ctx;

if (ndctl_new(&ctx) == 0)
        return ctx;
else
        return NULL;
```

LIBNVDIMM/LIBNDCTL: Bus

A bus has a 1:1 relationship with an NFIT. The current expectation for ACPI based systems is that there is only ever one platform-global NFIT. That said, it is trivial to register multiple NFITs, the specification does not preclude it. The infrastructure supports multiple busses and we use this capability to test multiple NFIT configurations in the unit test.

LIBNVDIMM: control class device in /sys/class

This character device accepts DSM messages to be passed to DIMM identified by its NFIT handle:

```
/sys/class/nd/ndctl0
|-- dev
|-- device -> ../../../ndbus0
|-- subsystem -> ../../../../../../../class/nd
```

LIBNVDIMM: bus

```
struct nvdimm_bus *nvdimm_bus_register(struct device *parent,
       struct nvdimm_bus_descriptor *nfit_desc);
```

```
/sys/devices/platform/nfit_test.0/ndbus0
|-- commands
|-- nd
|-- nfit
|-- nmem0
|-- nmem1
|-- nmem2
|-- nmem3
|-- power
|-- provider
|-- region0
|-- region1
|-- region2
|-- region3
|-- region4
|-- region5
|-- uevent
`-- wait_probe
```

LIBNDCTL: bus enumeration example

Find the bus handle that describes the bus from Example NVDIMM Platform:

```
static struct ndctl_bus *get_bus_by_provider(struct ndctl_ctx *ctx,
                const char *provider)
{
        struct ndctl_bus *bus;

        ndctl_bus_foreach(ctx, bus)
                if (strcmp(provider, ndctl_bus_get_provider(bus)) == 0)
                        return bus;

        return NULL;
}

bus = get_bus_by_provider(ctx, "nfit_test.0");
```

LIBNVDIMM/LIBNDCTL: DIMM (NMEM)

The DIMM device provides a character device for sending commands to hardware, and it is a container for LABELs. If the DIMM is defined by NFIT then an optional 'nfit' attribute sub-directory is available to add NFIT-specifics.

Note that the kernel device name for "DIMMs" is "nmemX". The NFIT describes these devices via "Memory Device to System Physical Address Range Mapping Structure", and there is no requirement that they actually be physical DIMMs, so we use a more generic name.

LIBNVDIMM: DIMM (NMEM)

```
struct nvdimm *nvdimm_create(struct nvdimm_bus *nvdimm_bus, void *provider_
↪data,
                const struct attribute_group **groups, unsigned long flags,
                unsigned long *dsm_mask);
```

```
/sys/devices/platform/nfit_test.0/ndbus0
|-- nmem0
|   |-- available_slots
|   |-- commands
|   |-- dev
|   |-- devtype
|   |-- driver -> ../../../../../bus/nd/drivers/nvdimm
|   |-- modalias
|   |-- nfit
|   |   |-- device
|   |   |-- format
|   |   |-- handle
|   |   |-- phys_id
|   |   |-- rev_id
|   |   |-- serial
|   |   `-- vendor
```

(continues on next page)

(continued from previous page)

```
|   |-- state
|   |-- subsystem -> ../../../../../bus/nd
|   `-- uevent
|-- nmem1
[..]
```

LIBNDCTL: DIMM enumeration example

Note, in this example we are assuming NFIT-defined DIMMs which are identified by an “nfit_handle” a 32-bit value where:

- Bit 3:0 DIMM number within the memory channel
- Bit 7:4 memory channel number
- Bit 11:8 memory controller ID
- Bit 15:12 socket ID (within scope of a Node controller if node controller is present)
- Bit 27:16 Node Controller ID
- Bit 31:28 Reserved

```
static struct ndctl_dimm *get_dimm_by_handle(struct ndctl_bus *bus,
       unsigned int handle)
{
        struct ndctl_dimm *dimm;

        ndctl_dimm_foreach(bus, dimm)
                if (ndctl_dimm_get_handle(dimm) == handle)
                        return dimm;

        return NULL;
}

#define DIMM_HANDLE(n, s, i, c, d) \
        (((n & 0xfff) << 16) | ((s & 0xf) << 12) | ((i & 0xf) << 8) \
         | ((c & 0xf) << 4) | (d & 0xf))

dimm = get_dimm_by_handle(bus, DIMM_HANDLE(0, 0, 0, 0, 0));
```

LIBNVDIMM/LIBNDCTL: Region

A generic REGION device is registered for each PMEM range or BLK-aperture set. Per the example there are 6 regions: 2 PMEM and 4 BLK-aperture sets on the “nfit_test.0” bus. The primary role of regions are to be a container of “mappings” . A mapping is a tuple of <DIMM, DPA-start-offset, length>.

LIBNVDIMM provides a built-in driver for these REGION devices. This driver is responsible for reconciling the aliased DPA mappings across all regions, parsing the LABEL, if present, and then emitting NAMESPACE devices with the resolved/exclusive DPA-boundaries for the nd_pmem or nd_blk device driver to consume.

In addition to the generic attributes of “mapping” s, “interleave_ways” and “size” the REGION device also exports some convenience attributes. “nstype” indicates the integer type of namespace-device this region emits, “devtype” duplicates the DEVTYPE variable stored by udev at the ‘add’ event, “modalias” duplicates the MODALIAS variable stored by udev at the ‘add’ event, and finally, the optional “spa_index” is provided in the case where the region is defined by a SPA.

LIBNVDIMM: region:

```
struct nd_region *nvdimm_pmem_region_create(struct nvdimm_bus *nvdimm_bus,
                struct nd_region_desc *ndr_desc);
struct nd_region *nvdimm_blk_region_create(struct nvdimm_bus *nvdimm_bus,
                struct nd_region_desc *ndr_desc);
```

```
/sys/devices/platform/nfit_test.0/ndbus0
|-- region0
|   |-- available_size
|   |-- btt0
|   |-- btt_seed
|   |-- devtype
|   |-- driver -> ../../../../../bus/nd/drivers/nd_region
|   |-- init_namespaces
|   |-- mapping0
|   |-- mapping1
|   |-- mappings
|   |-- modalias
|   |-- namespace0.0
|   |-- namespace_seed
|   |-- numa_node
|   |-- nfit
|   |   `-- spa_index
|   |-- nstype
|   |-- set_cookie
|   |-- size
|   |-- subsystem -> ../../../../../bus/nd
|   `-- uevent
|-- region1
[..]
```

LIBNDCTL: region enumeration example

Sample region retrieval routines based on NFIT-unique data like “spa_index” (interleave set id) for PMEM and “nfit_handle” (dimm id) for BLK:

```
static struct ndctl_region *get_pmem_region_by_spa_index(struct ndctl_bus␣
↪*bus,
                unsigned int spa_index)
{
        struct ndctl_region *region;

        ndctl_region_foreach(bus, region) {
                if (ndctl_region_get_type(region) != ND_DEVICE_REGION_PMEM)
                        continue;
                if (ndctl_region_get_spa_index(region) == spa_index)
```

(continues on next page)

(continued from previous page)

```
                        return region;
        }
        return NULL;
}

static struct ndctl_region *get_blk_region_by_dimm_handle(struct ndctl_bus␣
↪*bus,
                unsigned int handle)
{
        struct ndctl_region *region;

        ndctl_region_foreach(bus, region) {
                struct ndctl_mapping *map;

                if (ndctl_region_get_type(region) != ND_DEVICE_REGION_
↪BLOCK)
                        continue;
                ndctl_mapping_foreach(region, map) {
                        struct ndctl_dimm *dimm = ndctl_mapping_get_
↪dimm(map);

                        if (ndctl_dimm_get_handle(dimm) == handle)
                                return region;
                }
        }
        return NULL;
}
```

Why Not Encode the Region Type into the Region Name?

At first glance it seems since NFIT defines just PMEM and BLK interface types that we should simply name REGION devices with something derived from those type names. However, the ND subsystem explicitly keeps the REGION name generic and expects userspace to always consider the region-attributes for four reasons:

1. There are already more than two REGION and “namespace”types. For PMEM there are two subtypes. As mentioned previously we have PMEM where the constituent DIMM devices are known and anonymous PMEM. For BLK regions the NFIT specification already anticipates vendor specific implementations. The exact distinction of what a region contains is in the region-attributes not the region-name or the region-devtype.

2. A region with zero child-namespaces is a possible configuration. For example, the NFIT allows for a DCR to be published without a corresponding BLK-aperture. This equates to a DIMM that can only accept control/configuration messages, but no i/o through a descendant block device. Again, this “type” is advertised in the attributes (‘mappings’ == 0) and the name does not tell you much.

3. What if a third major interface type arises in the future? Outside of vendor specific implementations, it’s not difficult to envision a third class of interface type beyond BLK and PMEM. With a generic name for the REGION level of the device-hierarchy old userspace implementations can still make sense of new kernel advertised region-types. Userspace can always rely on the generic

region attributes like “mappings”, “size”, etc and the expected child devices named “namespace”. This generic format of the device-model hierarchy allows the LIBNVDIMM and LIBNDCTL implementations to be more uniform and future-proof.

4. There are more robust mechanisms for determining the major type of a region than a device name. See the next section, How Do I Determine the Major Type of a Region?

How Do I Determine the Major Type of a Region?

Outside of the blanket recommendation of “use libndctl”, or simply looking at the kernel header (/usr/include/linux/ndctl.h) to decode the “nstype” integer attribute, here are some other options.

1. module alias lookup

The whole point of region/namespace device type differentiation is to decide which block-device driver will attach to a given LIBNVDIMM namespace. One can simply use the modalias to lookup the resulting module. It’s important to note that this method is robust in the presence of a vendor-specific driver down the road. If a vendor-specific implementation wants to supplant the standard nd_blk driver it can with minimal impact to the rest of LIBNVDIMM.

In fact, a vendor may also want to have a vendor-specific region-driver (outside of nd_region). For example, if a vendor defined its own LABEL format it would need its own region driver to parse that LABEL and emit the resulting namespaces. The output from module resolution is more accurate than a region-name or region-devtype.

2. udev

The kernel “devtype” is registered in the udev database:

```
# udevadm info --path=/devices/platform/nfit_test.0/ndbus0/region0
P: /devices/platform/nfit_test.0/ndbus0/region0
E: DEVPATH=/devices/platform/nfit_test.0/ndbus0/region0
E: DEVTYPE=nd_pmem
E: MODALIAS=nd:t2
E: SUBSYSTEM=nd

# udevadm info --path=/devices/platform/nfit_test.0/ndbus0/region4
P: /devices/platform/nfit_test.0/ndbus0/region4
E: DEVPATH=/devices/platform/nfit_test.0/ndbus0/region4
E: DEVTYPE=nd_blk
E: MODALIAS=nd:t3
E: SUBSYSTEM=nd
```

…and is available as a region attribute, but keep in mind that the “devtype” does not indicate sub-type variations and scripts should really be understanding the other attributes.

3. type specific attributes

As it currently stands a BLK-aperture region will never have a "nfit/spa_index" attribute, but neither will a non-NFIT PMEM region. A BLK region with a "mappings" value of 0 is, as mentioned above, a DIMM that does not allow I/O. A PMEM region with a "mappings" value of zero is a simple system-physical-address range.

LIBNVDIMM/LIBNDCTL: Namespace

A REGION, after resolving DPA aliasing and LABEL specified boundaries, surfaces one or more "namespace" devices. The arrival of a "namespace" device currently triggers either the nd_blk or nd_pmem driver to load and register a disk/block device.

LIBNVDIMM: namespace

Here is a sample layout from the three major types of NAMESPACE where namespace0.0 represents DIMM-info-backed PMEM (note that it has a 'uuid' attribute), namespace2.0 represents a BLK namespace (note it has a 'sector_size' attribute) that, and namespace6.0 represents an anonymous PMEM namespace (note that has no 'uuid' attribute due to not support a LABEL):

```
/sys/devices/platform/nfit_test.0/ndbus0/region0/namespace0.0
|-- alt_name
|-- devtype
|-- dpa_extents
|-- force_raw
|-- modalias
|-- numa_node
|-- resource
|-- size
|-- subsystem -> ../../../../../../bus/nd
|-- type
|-- uevent
`-- uuid
/sys/devices/platform/nfit_test.0/ndbus0/region2/namespace2.0
|-- alt_name
|-- devtype
|-- dpa_extents
|-- force_raw
|-- modalias
|-- numa_node
|-- sector_size
|-- size
|-- subsystem -> ../../../../../../bus/nd
|-- type
|-- uevent
`-- uuid
/sys/devices/platform/nfit_test.1/ndbus1/region6/namespace6.0
|-- block
|   `-- pmem0
```

(continues on next page)

(continued from previous page)

```
|-- devtype
|-- driver -> ../../../../../../bus/nd/drivers/pmem
|-- force_raw
|-- modalias
|-- numa_node
|-- resource
|-- size
|-- subsystem -> ../../../../../../bus/nd
|-- type
`-- uevent
```

LIBNDCTL: namespace enumeration example

Namespaces are indexed relative to their parent region, example below. These indexes are mostly static from boot to boot, but subsystem makes no guarantees in this regard. For a static namespace identifier use its 'uuid' attribute.

```
static struct ndctl_namespace
*get_namespace_by_id(struct ndctl_region *region, unsigned int id)
{
        struct ndctl_namespace *ndns;

        ndctl_namespace_foreach(region, ndns)
                if (ndctl_namespace_get_id(ndns) == id)
                        return ndns;

        return NULL;
}
```

LIBNDCTL: namespace creation example

Idle namespaces are automatically created by the kernel if a given region has enough available capacity to create a new namespace. Namespace instantiation involves finding an idle namespace and configuring it. For the most part the setting of namespace attributes can occur in any order, the only constraint is that 'uuid' must be set before 'size'. This enables the kernel to track DPA allocations internally with a static identifier:

```
static int configure_namespace(struct ndctl_region *region,
                struct ndctl_namespace *ndns,
                struct namespace_parameters *parameters)
{
        char devname[50];

        snprintf(devname, sizeof(devname), "namespace%d.%d",
                        ndctl_region_get_id(region), paramaters->id);

        ndctl_namespace_set_alt_name(ndns, devname);
        /* 'uuid' must be set prior to setting size! */
        ndctl_namespace_set_uuid(ndns, paramaters->uuid);
        ndctl_namespace_set_size(ndns, paramaters->size);
```

(continues on next page)

(continued from previous page)

```
        /* unlike pmem namespaces, blk namespaces have a sector size */
        if (parameters->lbasize)
                ndctl_namespace_set_sector_size(ndns, parameters->lbasize);
        ndctl_namespace_enable(ndns);
}
```

Why the Term “namespace”?

1. Why not “volume” for instance? “volume” ran the risk of confusing ND (libnvdimm subsystem) to a volume manager like device-mapper.
2. The term originated to describe the sub-devices that can be created within a NVME controller (see the nvme specification: http://www.nvmexpress.org/specifications/), and NFIT namespaces are meant to parallel the capabilities and configurability of NVME-namespaces.

LIBNVDIMM/LIBNDCTL: Block Translation Table “btt”

A BTT (design document: http://pmem.io/2014/09/23/btt.html) is a stacked block device driver that fronts either the whole block device or a partition of a block device emitted by either a PMEM or BLK NAMESPACE.

LIBNVDIMM: btt layout

Every region will start out with at least one BTT device which is the seed device. To activate it set the “namespace”, “uuid”, and “sector_size” attributes and then bind the device to the nd_pmem or nd_blk driver depending on the region type:

```
/sys/devices/platform/nfit_test.1/ndbus0/region0/btt0/
|-- namespace
|-- delete
|-- devtype
|-- modalias
|-- numa_node
|-- sector_size
|-- subsystem -> ../../../../../bus/nd
|-- uevent
`-- uuid
```

LIBNDCTL: btt creation example

Similar to namespaces an idle BTT device is automatically created per region. Each time this “seed” btt device is configured and enabled a new seed is created. Creating a BTT configuration involves two steps of finding and idle BTT and assigning it to consume a PMEM or BLK namespace:

```
static struct ndctl_btt *get_idle_btt(struct ndctl_region *region)
{
        struct ndctl_btt *btt;

        ndctl_btt_foreach(region, btt)
                if (!ndctl_btt_is_enabled(btt)
                                && !ndctl_btt_is_configured(btt))
                        return btt;

        return NULL;
}

static int configure_btt(struct ndctl_region *region,
                struct btt_parameters *parameters)
{
        btt = get_idle_btt(region);

        ndctl_btt_set_uuid(btt, parameters->uuid);
        ndctl_btt_set_sector_size(btt, parameters->sector_size);
        ndctl_btt_set_namespace(btt, parameters->ndns);
        /* turn off raw mode device */
        ndctl_namespace_disable(parameters->ndns);
        /* turn on btt access */
        ndctl_btt_enable(btt);
}
```

Once instantiated a new inactive btt seed device will appear underneath the region.

Once a “namespace” is removed from a BTT that instance of the BTT device will be deleted or otherwise reset to default values. This deletion is only at the device model level. In order to destroy a BTT the “info block” needs to be destroyed. Note, that to destroy a BTT the media needs to be written in raw mode. By default, the kernel will autodetect the presence of a BTT and disable raw mode. This autodetect behavior can be suppressed by enabling raw mode for the namespace via the ndctl_namespace_set_raw_mode() API.

Summary LIBNDCTL Diagram

For the given example above, here is the view of the objects as seen by the LIBNDCTL API:

```
            +---+
            |CTX|    +---------+   +--------------+  +---------------+
            +-+-+  +-> REGION0 +---> NAMESPACE0.0 +--> PMEM8 "pm0.0" |
              |    | +---------+   +--------------+  +---------------+
+-------+     |    | +---------+   +--------------+  +---------------+
| DIMM0 <-+   |    +-> REGION1 +---> NAMESPACE1.0 +--> PMEM6 "pm1.0" |
+-------+ |   |    | +---------+   +--------------+  +---------------+
| DIMM1 <-+ +-v--+ | +---------+   +--------------+  +---------------+
+-------+ +-+BUS0+---> REGION2 +-+-> NAMESPACE2.0 +--> ND6  "blk2.0" |
| DIMM2 <-+ +----+ | +---------+ | +--------------+  +----------------------
↪-+
+-------+ |        |             +-> NAMESPACE2.1 +--> ND5  "blk2.1" |␣
↪BTT2 |
```

(continues on next page)

(continued from previous page)

```
| DIMM3 <-+        |               +--------------+  +---------------------
↪-+
+-------+          | +---------+   +--------------+  +---------------+
                   +-> REGION3 +-+-> NAMESPACE3.0 +--> ND4  "blk3.0" |
                   | +---------+ | +--------------+  +---------------------
↪-+
                   |             +-> NAMESPACE3.1 +--> ND3  "blk3.1" |␣
↪BTT1 |
                   |               +--------------+  +---------------------
↪-+
                   | +---------+   +--------------+  +---------------+
                   +-> REGION4 +---> NAMESPACE4.0 +--> ND2  "blk4.0" |
                   | +---------+   +--------------+  +---------------+
                   | +---------+   +--------------+  +---------------------
↪-+
                   +-> REGION5 +---> NAMESPACE5.0 +--> ND1  "blk5.0" |␣
↪BTT0 |
                     +---------+   +--------------+  +---------------+-----
↪-+
```

42.2 BTT - Block Translation Table

42.2.1 1. Introduction

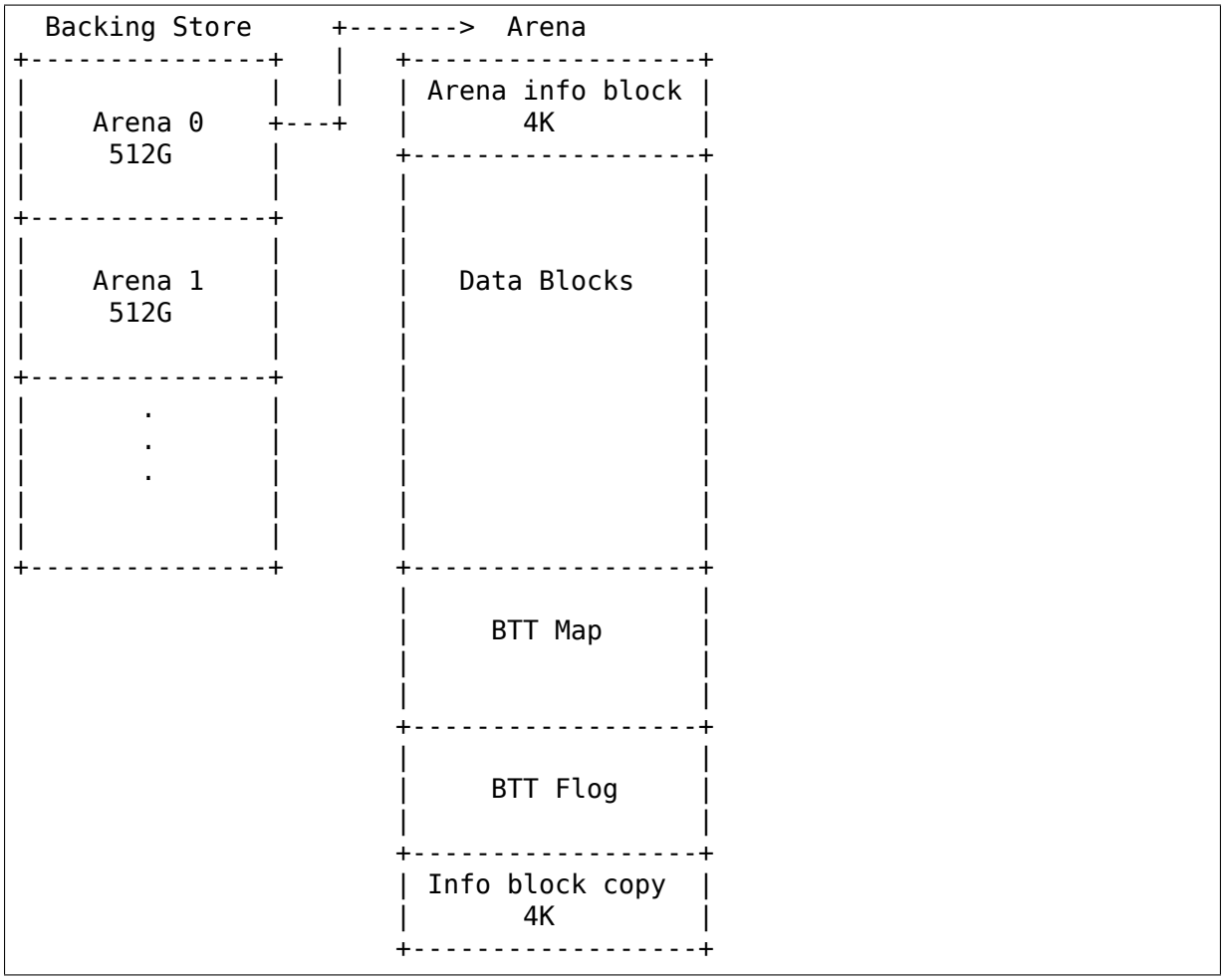
Persistent memory based storage is able to perform IO at byte (or more accurately, cache line) granularity. However, we often want to expose such storage as traditional block devices. The block drivers for persistent memory will do exactly this. However, they do not provide any atomicity guarantees. Traditional SSDs typically provide protection against torn sectors in hardware, using stored energy in capacitors to complete in-flight block writes, or perhaps in firmware. We don't have this luxury with persistent memory - if a write is in progress, and we experience a power failure, the block will contain a mix of old and new data. Applications may not be prepared to handle such a scenario.

The Block Translation Table (BTT) provides atomic sector update semantics for persistent memory devices, so that applications that rely on sector writes not being torn can continue to do so. The BTT manifests itself as a stacked block device, and reserves a portion of the underlying storage for its metadata. At the heart of it, is an indirection table that re-maps all the blocks on the volume. It can be thought of as an extremely simple file system that only provides atomic sector updates.

42.2.2 2. Static Layout

The underlying storage on which a BTT can be laid out is not limited in any way. The BTT, however, splits the available space into chunks of up to 512 GiB, called "Arenas".

Each arena follows the same layout for its metadata, and all references in an arena are internal to it (with the exception of one field that points to the next arena). The following depicts the "On-disk" metadata layout:



42.2.3 3. Theory of Operation

a. The BTT Map

The map is a simple lookup/indirection table that maps an LBA to an internal block. Each map entry is 32 bits. The two most significant bits are special flags, and the remaining form the internal block number.

Bit	Description

31 - 30	Error and Zero flags - Used in the following way:
	`== == ␣ ↪==========`
	`31 30  Description`
	`== == ␣ ↪==========`
	`0  0   Initial state. Reads return␣ ↪zeroes; Premap = Postmap`
	`0  1   Zero state: Reads return␣ ↪zeroes`
	`1  0   Error state: Reads fail;␣ ↪Writes clear 'E' bit`
	`1  1   Normal Block – has valid␣ ↪postmap`
	`== == ␣ ↪==========`

29 - 0	Mappings to internal ‘postmap’ blocks

Some of the terminology that will be subsequently used:

External LBA	LBA as made visible to upper layers.
ABA	Arena Block Address - Block offset/number within an arena
Premap ABA	The block offset into an arena, which was decided upon by range checking the External LBA
Postmap ABA	The block number in the “Data Blocks” area obtained after indirection from the map
nfree	The number of free blocks that are maintained at any given time. This is the number of concurrent writes that can happen to the arena.

For example, after adding a BTT, we surface a disk of 1024G. We get a read for the external LBA at 768G. This falls into the second arena, and of the 512G worth of blocks that this arena contributes, this block is at 256G. Thus, the premap ABA is 256G. We now refer to the map, and find out the mapping for block ‘X’ (256G) points to block ‘Y’, say ‘64’. Thus the postmap ABA is 64.

b. The BTT Flog

The BTT provides sector atomicity by making every write an “allocating write”, i.e. Every write goes to a “free” block. A running list of free blocks is maintained in the form of the BTT flog. ‘Flog’ is a combination of the words “free list” and “log”. The flog contains ‘nfree’ entries, and an entry contains:

lba	The premap ABA that is being written to
old_map	The old postmap ABA - after 'this' write completes, this will be a free block.
new_map	The new postmap ABA. The map will up updated to reflect this lba->postmap_aba mapping, but we log it here in case we have to recover.
seq	Sequence number to mark which of the 2 sections of this flog entry is valid/newest. It cycles between 01->10->11->01 (binary) under normal operation, with 00 indicating an uninitialized state.
lba'	alternate lba entry
old_map'	alternate old postmap entry
new_map'	alternate new postmap entry
seq'	alternate sequence number.

Each of the above fields is 32-bit, making one entry 32 bytes. Entries are also padded to 64 bytes to avoid cache line sharing or aliasing. Flog updates are done such that for any entry being written, it: a. overwrites the 'old' section in the entry based on sequence numbers b. writes the 'new' section such that the sequence number is written last.

c. The concept of lanes

While 'nfree' describes the number of concurrent IOs an arena can process concurrently, 'nlanes' is the number of IOs the BTT device as a whole can process:

```
nlanes = min(nfree, num_cpus)
```

A lane number is obtained at the start of any IO, and is used for indexing into all the on-disk and in-memory data structures for the duration of the IO. If there are more CPUs than the max number of available lanes, than lanes are protected by spinlocks.

d. In-memory data structure: Read Tracking Table (RTT)

Consider a case where we have two threads, one doing reads and the other, writes. We can hit a condition where the writer thread grabs a free block to do a new IO, but the (slow) reader thread is still reading from it. In other words, the reader consulted a map entry, and started reading the corresponding block. A writer started writing to the same external LBA, and finished the write updating the map for that external LBA to point to its new postmap ABA. At this point the internal, postmap block that the reader is (still) reading has been inserted into the list of free blocks. If another write comes in for the same LBA, it can grab this free block, and start writing to it, causing the reader to read incorrect data. To prevent this, we introduce the RTT.

The RTT is a simple, per arena table with 'nfree' entries. Every reader inserts into rtt[lane_number], the postmap ABA it is reading, and clears it after the read is complete. Every writer thread, after grabbing a free block, checks the RTT for its presence. If the postmap free block is in the RTT, it waits till the reader clears the RTT entry, and only then starts writing to it.

e. In-memory data structure: map locks

Consider a case where two writer threads are writing to the same LBA. There can be a race in the following sequence of steps:

```
free[lane] = map[premap_aba]
map[premap_aba] = postmap_aba
```

Both threads can update their respective free[lane] with the same old, freed postmap_aba. This has made the layout inconsistent by losing a free entry, and at the same time, duplicating another free entry for two lanes.

To solve this, we could have a single map lock (per arena) that has to be taken before performing the above sequence, but we feel that could be too contentious. Instead we use an array of (nfree) map_locks that is indexed by (premap_aba modulo nfree).

f. Reconstruction from the Flog

On startup, we analyze the BTT flog to create our list of free blocks. We walk through all the entries, and for each lane, of the set of two possible 'sections', we always look at the most recent one only (based on the sequence number). The reconstruction rules/steps are simple:

- Read map[log_entry.lba].
- If log_entry.new matches the map entry, then log_entry.old is free.
- If log_entry.new does not match the map entry, then log_entry.new is free. (This case can only be caused by power-fails/unsafe shutdowns)

g. Summarizing - Read and Write flows

Read:

1. Convert external LBA to arena number + pre-map ABA
2. Get a lane (and take lane_lock)
3. Read map to get the entry for this pre-map ABA
4. Enter post-map ABA into RTT[lane]
5. If TRIM flag set in map, return zeroes, and end IO (go to step 8)
6. If ERROR flag set in map, end IO with EIO (go to step 8)
7. Read data from this block
8. Remove post-map ABA entry from RTT[lane]
9. Release lane (and lane_lock)

Write:

1. Convert external LBA to Arena number + pre-map ABA
2. Get a lane (and take lane_lock)

3. Use lane to index into in-memory free list and obtain a new block, next flog index, next sequence number
4. Scan the RTT to check if free block is present, and spin/wait if it is.
5. Write data to this free block
6. Read map to get the existing post-map ABA entry for this pre-map ABA
7. Write flog entry: [premap_aba / old postmap_aba / new postmap_aba / seq_num]
8. Write new post-map ABA into map.
9. Write old post-map entry into the free list
10. Calculate next sequence number and write into the free list entry
11. Release lane (and lane_lock)

42.2.4 4. Error Handling

An arena would be in an error state if any of the metadata is corrupted irrecoverably, either due to a bug or a media error. The following conditions indicate an error:

- Info block checksum does not match (and recovering from the copy also fails)
- All internal available blocks are not uniquely and entirely addressed by the sum of mapped blocks and free blocks (from the BTT flog).
- Rebuilding free list from the flog reveals missing/duplicate/impossible entries
- A map entry is out of bounds

If any of these error conditions are encountered, the arena is put into a read only state using a flag in the info block.

42.2.5 5. Usage

The BTT can be set up on any disk (namespace) exposed by the libnvdimm subsystem (pmem, or blk mode). The easiest way to set up such a namespace is using the 'ndctl' utility [1]:

For example, the ndctl command line to setup a btt with a 4k sector size is:

```
ndctl create-namespace -f -e namespace0.0 -m sector -l 4k
```

See ndctl create-namespace –help for more options.

[1]: https://github.com/pmem/ndctl

42.3 NVDIMM Security

42.3.1 1. Introduction

With the introduction of Intel Device Specific Methods (DSM) v1.8 specification [1], security DSMs are introduced. The spec added the following security DSMs: “get security state”, “set passphrase”, “disable passphrase”, “unlock unit”, “freeze lock”, “secure erase”, and “overwrite”. A security_ops data structure has been added to struct dimm in order to support the security operations and generic APIs are exposed to allow vendor neutral operations.

42.3.2 2. Sysfs Interface

The “security” sysfs attribute is provided in the nvdimm sysfs directory. For example: /sys/devices/LNXSYSTM:00/LNXSYBUS:00/ACPI0012:00/ndbus0/nmem0/security

The “show” attribute of that attribute will display the security state for that DIMM. The following states are available: disabled, unlocked, locked, frozen, and overwrite. If security is not supported, the sysfs attribute will not be visible.

The “store” attribute takes several commands when it is being written to in order to support some of the security functionalities: update <old_keyid> <new_keyid> - enable or update passphrase. disable <keyid> - disable enabled security and remove key. freeze - freeze changing of security states. erase <keyid> - delete existing user encryption key. overwrite <keyid> - wipe the entire nvdimm. master_update <keyid> <new_keyid> - enable or update master passphrase. master_erase <keyid> - delete existing user encryption key.

42.3.3 3. Key Management

The key is associated to the payload by the DIMM id. For example: # cat /sys/devices/LNXSYSTM:00/LNXSYBUS:00/ACPI0012:00/ndbus0/nmem0/nfit/id 8089-a2-1740-00000133 The DIMM id would be provided along with the key payload (passphrase) to the kernel.

The security keys are managed on the basis of a single key per DIMM. The key “passphrase” is expected to be 32bytes long. This is similar to the ATA security specification [2]. A key is initially acquired via the request_key() kernel API call during nvdimm unlock. It is up to the user to make sure that all the keys are in the kernel user keyring for unlock.

A nvdimm encrypted-key of format enc32 has the description format of: nvdimm:<bus-provider-specific-unique-id>

See file `Documentation/security/keys/trusted-encrypted.rst` for creating encrypted-keys of enc32 format. TPM usage with a master trusted key is preferred for sealing the encrypted-keys.

42.3.4 4. Unlocking

When the DIMMs are being enumerated by the kernel, the kernel will attempt to retrieve the key from the kernel user keyring. This is the only time a locked DIMM can be unlocked. Once unlocked, the DIMM will remain unlocked until reboot. Typically an entity (i.e. shell script) will inject all the relevant encrypted-keys into the kernel user keyring during the initramfs phase. This provides the unlock function access to all the related keys that contain the passphrase for the respective nvdimms. It is also recommended that the keys are injected before libnvdimm is loaded by modprobe.

42.3.5 5. Update

When doing an update, it is expected that the existing key is removed from the kernel user keyring and reinjected as different (old) key. It' s irrelevant what the key description is for the old key since we are only interested in the keyid when doing the update operation. It is also expected that the new key is injected with the description format described from earlier in this document. The update command written to the sysfs attribute will be with the format: update <old keyid> <new keyid>

If there is no old keyid due to a security enabling, then a 0 should be passed in.

42.3.6 6. Freeze

The freeze operation does not require any keys. The security config can be frozen by a user with root privelege.

42.3.7 7. Disable

The security disable command format is: disable <keyid>

An key with the current passphrase payload that is tied to the nvdimm should be in the kernel user keyring.

42.3.8 8. Secure Erase

The command format for doing a secure erase is: erase <keyid>

An key with the current passphrase payload that is tied to the nvdimm should be in the kernel user keyring.

42.3.9 9. Overwrite

The command format for doing an overwrite is: overwrite <keyid>

Overwrite can be done without a key if security is not enabled. A key serial of 0 can be passed in to indicate no key.

The sysfs attribute "security" can be polled to wait on overwrite completion. Overwrite can last tens of minutes or more depending on nvdimm size.

An encrypted-key with the current user passphrase that is tied to the nvdimm should be injected and its keyid should be passed in via sysfs.

42.3.10 10. Master Update

The command format for doing a master update is: update <old keyid> <new keyid>

The operating mechanism for master update is identical to update except the master passphrase key is passed to the kernel. The master passphrase key is just another encrypted-key.

This command is only available when security is disabled.

42.3.11 11. Master Erase

The command format for doing a master erase is: master_erase <current keyid>

This command has the same operating mechanism as erase except the master passphrase key is passed to the kernel. The master passphrase key is just another encrypted-key.

This command is only available when the master security is enabled, indicated by the extended security status.

[1]: http://pmem.io/documents/NVDIMM_DSM_Interface-V1.8.pdf

[2]: http://www.t13.org/documents/UploadedDocuments/docs2006/e05179r4-ACS-SecurityClarifications.pdf

CHAPTER

FORTYTHREE

W1: DALLAS' 1-WIRE BUS

Author David Fries

43.1 W1 API internal to the kernel

43.1.1 include/linux/w1.h

W1 kernel API functions.

struct **w1_reg_num**
broken out slave device id

Definition

```
struct w1_reg_num {
#if defined(__LITTLE_ENDIAN_BITFIELD);
  __u64 family:8,id:48, crc:8;
#elif defined(__BIG_ENDIAN_BITFIELD);
  __u64 crc:8,id:48, family:8;
#else;
#error "Please fix <asm/byteorder.h>";
#endif;
};
```

Members

family identifies the type of device

id along with family is the unique device id

crc checksum of the other bytes

crc checksum of the other bytes

id along with family is the unique device id

family identifies the type of device

struct **w1_slave**
holds a single slave device on the bus

Definition

```
struct w1_slave {
  struct module           *owner;
  unsigned char           name[W1_MAXNAMELEN];
  struct list_head        w1_slave_entry;
  struct w1_reg_num       reg_num;
  atomic_t refcnt;
  int ttl;
  unsigned long           flags;
  struct w1_master        *master;
  struct w1_family        *family;
  void *family_data;
  struct device           dev;
  struct device           *hwmon;
};
```

Members

owner Points to the one wire “wire” kernel module.

name Device id is ascii.

w1_slave_entry data for the linked list

reg_num the slave id in binary

refcnt reference count, delete when 0

ttl decrement per search this slave isn’t found, deatch at 0

flags bit flags for W1_SLAVE_ACTIVE W1_SLAVE_DETACH

master bus which this slave is on

family module for device family type

family_data pointer for use by the family module

dev kernel device identifier

hwmon pointer to hwmon device

struct **w1_bus_master**
operations available on a bus master

Definition

```
struct w1_bus_master {
  void *data;
  u8 (*read_bit)(void *);
  void (*write_bit)(void *, u8);
  u8 (*touch_bit)(void *, u8);
  u8 (*read_byte)(void *);
  void (*write_byte)(void *, u8);
  u8 (*read_block)(void *, u8 *, int);
  void (*write_block)(void *, const u8 *, int);
  u8 (*triplet)(void *, u8);
  u8 (*reset_bus)(void *);
  u8 (*set_pullup)(void *, int);
  void (*search)(void *, struct w1_master *, u8, w1_slave_found_callback);
  char *dev_id;
};
```

Members

data the first parameter in all the functions below

read_bit Sample the line level **return** the level read (0 or 1)

write_bit Sets the line level

touch_bit the lowest-level function for devices that really support the 1-wire protocol. touch_bit(0) = write-0 cycle touch_bit(1) = write-1 / read cycle **return** the bit read (0 or 1)

read_byte Reads a bytes. Same as 8 touch_bit(1) calls. **return** the byte read

write_byte Writes a byte. Same as 8 touch_bit(x) calls.

read_block Same as a series of read_byte() calls **return** the number of bytes read

write_block Same as a series of write_byte() calls

triplet Combines two reads and a smart write for ROM searches **return** bit0=Id bit1=comp_id bit2=dir_taken

reset_bus long write-0 with a read for the presence pulse detection **return** -1=Error, 0=Device present, 1=No device present

set_pullup Put out a strong pull-up pulse of the specified duration. **return** -1=Error, 0=completed

search Really nice hardware can handles the different types of ROM search w1_master* is passed to the slave found callback. u8 is search_type, W1_SEARCH or W1_ALARM_SEARCH

dev_id Optional device id string, which w1 slaves could use for creating names, which then give a connection to the w1 master

Note

read_bit and write_bit are very low level functions and should only be used with hardware that doesn't really support 1-wire operations, like a parallel/serial port. Either define read_bit and write_bit OR define, at minimum, touch_bit and reset_bus.

enum **w1_master_flags**
bitfields used in w1_master.flags

Constants

W1_ABORT_SEARCH abort searching early on shutdown

W1_WARN_MAX_COUNT limit warning when the maximum count is reached

struct **w1_master**
one per bus master

Definition

```
struct w1_master {
  struct list_head        w1_master_entry;
  struct module           *owner;
  unsigned char           name[W1_MAXNAMELEN];
  struct mutex            list_mutex;
```

(continues on next page)

(continued from previous page)

```
  struct list_head        slist;
  struct list_head        async_list;
  int max_slave_count, slave_count;
  unsigned long           attempts;
  int slave_ttl;
  int initialized;
  u32 id;
  int search_count;
  u64 search_id;
  atomic_t refcnt;
  void *priv;
  int enable_pullup;
  int pullup_duration;
  long flags;
  struct task_struct      *thread;
  struct mutex            mutex;
  struct mutex            bus_mutex;
  struct device_driver    *driver;
  struct device           dev;
  struct w1_bus_master    *bus_master;
  u32 seq;
};
```

Members

w1_master_entry master linked list

owner module owner

name dynamically allocate bus name

list_mutex protect slist and async_list

slist linked list of slaves

async_list linked list of netlink commands to execute

max_slave_count maximum number of slaves to search for at a time

slave_count current number of slaves known

attempts number of searches ran

slave_ttl number of searches before a slave is timed out

initialized prevent init/removal race conditions

id w1 bus number

search_count number of automatic searches to run, -1 unlimited

search_id allows continuing a search

refcnt reference count

priv private data storage

enable_pullup allows a strong pullup

pullup_duration time for the next strong pullup

flags one of w1_master_flags

thread thread for bus search and netlink commands

mutex protect most of w1_master

bus_mutex pretect concurrent bus access

driver sysfs driver

dev sysfs device

bus_master io operations available

seq sequence number used for netlink broadcasts

struct **w1_family_ops**
operations for a family type

Definition

```
struct w1_family_ops {
  int (*add_slave)(struct w1_slave *sl);
  void (*remove_slave)(struct w1_slave *sl);
  const struct attribute_group **groups;
  const struct hwmon_chip_info *chip_info;
};
```

Members

add_slave add_slave

remove_slave remove_slave

groups sysfs group

chip_info pointer to struct hwmon_chip_info

struct **w1_family**
reference counted family structure.

Definition

```
struct w1_family {
  struct list_head        family_entry;
  u8 fid;
  struct w1_family_ops    *fops;
  const struct of_device_id *of_match_table;
  atomic_t refcnt;
};
```

Members

family_entry family linked list

fid 8 bit family identifier

fops operations for this family

of_match_table open firmware match table

refcnt reference counter

module_w1_family(__w1_family)
Helper macro for registering a 1-Wire families

Parameters

__w1_family w1_family struct

Description

Helper macro for 1-Wire families which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init()` and `module_exit()`

43.1.2 drivers/w1/w1.c

W1 core functions.

void **w1_search**(struct w1_master * dev, u8 search_type, w1_slave_found_callback cb)
Performs a ROM Search & registers any devices found.

Parameters

struct w1_master * dev The master device to search

u8 search_type W1_SEARCH to search all devices, or W1_ALARM_SEARCH to return only devices in the alarmed state

w1_slave_found_callback cb Function to call when a device is found

Description

The 1-wire search is a simple binary tree search. For each bit of the address, we read two bits and write one bit. The bit written will put to sleep all devies that don't match that bit. When the two reads differ, the direction choice is obvious. When both bits are 0, we must choose a path to take. When we can scan all 64 bits without having to choose a path, we are done.

See "Application note 187 1-wire search algorithm" at www.maxim-ic.com

int **w1_process_callbacks**(struct w1_master * dev)
execute each dev->async_list callback entry

Parameters

struct w1_master * dev w1_master device

Description

The w1 master list_mutex must be held.

Return

1 if there were commands to executed 0 otherwise

43.1.3 drivers/w1/w1_family.c

Allows registering device family operations.

int **w1_register_family**(struct w1_family * newf)
register a device family driver

Parameters

struct w1_family * newf family to register

void **w1_unregister_family**(struct w1_family * fent)
unregister a device family driver

Parameters

struct w1_family * fent family to unregister

43.1.4 drivers/w1/w1_internal.h

W1 internal initialization for master devices.

struct **w1_async_cmd**
execute callback from the w1_process kthread

Definition

```
struct w1_async_cmd {
  struct list_head        async_entry;
  void (*cb)(struct w1_master *dev, struct w1_async_cmd *async_cmd);
};
```

Members

async_entry link entry

cb callback function, must list_del and destroy this list before returning

Description

When inserted into the w1_master async_list, w1_process will execute the callback. Embed this into the structure with the command details.

43.1.5 drivers/w1/w1_int.c

W1 internal initialization for master devices.

int **w1_add_master_device**(struct w1_bus_master * master)
registers a new master device

Parameters

struct w1_bus_master * master master bus device to register

void **w1_remove_master_device**(struct w1_bus_master * bm)
unregister a master device

Parameters

struct w1_bus_master * bm master bus device to remove

43.1.6 drivers/w1/w1_netlink.h

W1 external netlink API structures and commands.

enum **w1_cn_msg_flags**
bitfield flags for struct cn_msg.flags

Constants

W1_CN_BUNDLE Request bundling replies into fewer messagse. Be prepared to handle multiple struct cn_msg, struct w1_netlink_msg, and struct w1_netlink_cmd in one packet.

enum **w1_netlink_message_types**
message type

Constants

W1_SLAVE_ADD notification that a slave device was added

W1_SLAVE_REMOVE notification that a slave device was removed

W1_MASTER_ADD notification that a new bus master was added

W1_MASTER_REMOVE notification that a bus masterwas removed

W1_MASTER_CMD initiate operations on a specific master

W1_SLAVE_CMD sends reset, selects the slave, then does a read/write/touch operation

W1_LIST_MASTERS used to determine the bus master identifiers

struct **w1_netlink_msg**
holds w1 message type, id, and result

Definition

```
struct w1_netlink_msg {
  __u8 type;
  __u8 status;
  __u16 len;
  union {
    __u8 id[8];
    struct w1_mst {
      __u32 id;
      __u32 res;
    } mst;
  } id;
  __u8 data[];
};
```

Members

type one of enum w1_netlink_message_types

status kernel feedback for success 0 or errno failure value

len length of data following w1_netlink_msg

id union holding bus master id (msg.id) and slave device id (id[8]).

id.id Slave ID (8 bytes)

id.mst bus master identification

id.mst.id bus master ID

id.mst.res bus master reserved

data start address of any following data

Description

The base message structure for w1 messages over netlink. The netlink connector data sequence is, struct nlmsghdr, struct cn_msg, then one or more struct w1_netlink_msg (each with optional data).

enum **w1_commands**
commands available for master or slave operations

Constants

W1_CMD_READ read len bytes

W1_CMD_WRITE write len bytes

W1_CMD_SEARCH initiate a standard search, returns only the slave devices found during that search

W1_CMD_ALARM_SEARCH search for devices that are currently alarming

W1_CMD_TOUCH Touches a series of bytes.

W1_CMD_RESET sends a bus reset on the given master

W1_CMD_SLAVE_ADD adds a slave to the given master, 8 byte slave id at data[0]

W1_CMD_SLAVE_REMOVE removes a slave to the given master, 8 byte slave id at data[0]

W1_CMD_LIST_SLAVES list of slaves registered on this master

W1_CMD_MAX number of available commands

struct **w1_netlink_cmd**
holds the command and data

Definition

```
struct w1_netlink_cmd {
  __u8 cmd;
  __u8 res;
  __u16 len;
  __u8 data[];
};
```

Members

cmd one of enum w1_commands

res reserved

len length of data following w1_netlink_cmd

data start address of any following data

Description

One or more struct w1_netlink_cmd is placed starting at w1_netlink_msg.data each with optional data.

43.1.7 drivers/w1/w1_io.c

W1 input/output.

u8 **w1_touch_bit**(struct w1_master * dev, int bit)
Generates a write-0 or write-1 cycle and samples the level.

Parameters

struct w1_master * dev the master device

int bit 0 - write a 0, 1 - write a 0 read the level

void **w1_write_8**(struct w1_master * dev, u8 byte)
Writes 8 bits.

Parameters

struct w1_master * dev the master device

u8 byte the byte to write

u8 **w1_triplet**(struct w1_master * dev, int bdir)

- Does a triplet - used for searching ROM addresses.

Parameters

struct w1_master * dev the master device

int bdir the bit to write if both id_bit and comp_bit are 0

Description

Return bits: bit 0 = id_bit bit 1 = comp_bit bit 2 = dir_taken

If both bits 0 & 1 are set, the search should be restarted.

Return

bit fields - see above

u8 **w1_read_8**(struct w1_master * dev)
Reads 8 bits.

Parameters

struct w1_master * dev the master device

Return

the byte read

void **w1_write_block**(struct w1_master * dev, const u8 * buf, int len)
Writes a series of bytes.

Parameters

struct w1_master * dev the master device

const u8 * buf pointer to the data to write

int len the number of bytes to write

void **w1_touch_block**(struct w1_master * dev, u8 * buf, int len)
Touches a series of bytes.

Parameters

struct w1_master * dev the master device

u8 * buf pointer to the data to write

int len the number of bytes to write

u8 **w1_read_block**(struct w1_master * dev, u8 * buf, int len)
Reads a series of bytes.

Parameters

struct w1_master * dev the master device

u8 * buf pointer to the buffer to fill

int len the number of bytes to read

Return

the number of bytes read

int **w1_reset_bus**(struct w1_master * dev)
Issues a reset bus sequence.

Parameters

struct w1_master * dev the master device

Return

0=Device present, 1=No device present or error

int **w1_reset_select_slave**(struct w1_slave * sl)
reset and select a slave

Parameters

struct w1_slave * sl the slave to select

Description

Resets the bus and then selects the slave by sending either a skip rom or a rom match. A skip rom is issued if there is only one device registered on the bus. The w1 master lock must be held.

Return

0=success, anything else=error

int **w1_reset_resume_command**(struct w1_master * dev)
resume instead of another match ROM

Parameters

struct w1_master * dev the master device

Description

When the workflow with a slave amongst many requires several successive commands a reset between each, this function is similar to doing a reset then a match ROM for the last matched ROM. The advantage being that the matched ROM step is skipped in favor of the resume command. The slave must support the command of course.

If the bus has only one slave, traditionnaly the match ROM is skipped and a “SKIP ROM” is done for efficiency. On multi-slave busses, this doesn’t work of course, but the resume command is the next best thing.

The w1 master lock must be held.

void **w1_next_pullup**(struct w1_master * dev, int delay)
register for a strong pullup

Parameters

struct w1_master * dev the master device

int delay time in milliseconds

Description

Put out a strong pull-up of the specified duration after the next write operation. Not all hardware supports strong pullups. Hardware that doesn’t support strong pullups will sleep for the given time after the write operation without a strong pullup. This is a one shot request for the next write, specifying zero will clear a previous request. The w1 master lock must be held.

Return

0=success, anything else=error

void **w1_write_bit**(struct w1_master * dev, int bit)
Generates a write-0 or write-1 cycle.

Parameters

struct w1_master * dev the master device

int bit bit to write

Description

Only call if dev->bus_master->touch_bit is NULL

void **w1_pre_write**(struct w1_master * dev)
pre-write operations

Parameters

struct w1_master * dev the master device

Description

Pre-write operation, currently only supporting strong pullups. Program the hardware for a strong pullup, if one has been requested and the hardware supports it.

void **w1_post_write**(struct w1_master * dev)
post-write options

Parameters

struct w1_master * dev the master device

Description

Post-write operation, currently only supporting strong pullups. If a strong pullup was requested, clear it if the hardware supports them, or execute the delay otherwise, in either case clear the request.

u8 **w1_read_bit**(struct w1_master * dev)
Generates a write-1 cycle and samples the level.

Parameters

struct w1_master * dev the master device

Description

Only call if dev->bus_master->touch_bit is NULL

CHAPTER

FORTYFOUR

THE LINUX RAPIDIO SUBSYSTEM

44.1 Introduction

The RapidIO standard is a packet-based fabric interconnect standard designed for use in embedded systems. Development of the RapidIO standard is directed by the RapidIO Trade Association (RTA). The current version of the RapidIO specification is publicly available for download from the RTA web-site [1].

This document describes the basics of the Linux RapidIO subsystem and provides information on its major components.

44.1.1 1 Overview

Because the RapidIO subsystem follows the Linux device model it is integrated into the kernel similarly to other buses by defining RapidIO-specific device and bus types and registering them within the device model.

The Linux RapidIO subsystem is architecture independent and therefore defines architecture-specific interfaces that provide support for common RapidIO subsystem operations.

44.1.2 2. Core Components

A typical RapidIO network is a combination of endpoints and switches. Each of these components is represented in the subsystem by an associated data structure. The core logical components of the RapidIO subsystem are defined in include/linux/rio.h file.

2.1 Master Port

A master port (or mport) is a RapidIO interface controller that is local to the processor executing the Linux code. A master port generates and receives RapidIO packets (transactions). In the RapidIO subsystem each master port is represented by a rio_mport data structure. This structure contains master port specific resources such as mailboxes and doorbells. The rio_mport also includes a unique host device ID that is valid when a master port is configured as an enumerating host.

RapidIO master ports are serviced by subsystem specific mport device drivers that provide functionality defined for this subsystem. To provide a hardware independent interface for RapidIO subsystem operations, rio_mport structure includes rio_ops data structure which contains pointers to hardware specific implementations of RapidIO functions.

2.2 Device

A RapidIO device is any endpoint (other than mport) or switch in the network. All devices are presented in the RapidIO subsystem by corresponding rio_dev data structure. Devices form one global device list and per-network device lists (depending on number of available mports and networks).

2.3 Switch

A RapidIO switch is a special class of device that routes packets between its ports towards their final destination. The packet destination port within a switch is defined by an internal routing table. A switch is presented in the RapidIO subsystem by rio_dev data structure expanded by additional rio_switch data structure, which contains switch specific information such as copy of the routing table and pointers to switch specific functions.

The RapidIO subsystem defines the format and initialization method for subsystem specific switch drivers that are designed to provide hardware-specific implementation of common switch management routines.

2.4 Network

A RapidIO network is a combination of interconnected endpoint and switch devices. Each RapidIO network known to the system is represented by corresponding rio_net data structure. This structure includes lists of all devices and local master ports that form the same network. It also contains a pointer to the default master port that is used to communicate with devices within the network.

2.5 Device Drivers

RapidIO device-specific drivers follow Linux Kernel Driver Model and are intended to support specific RapidIO devices attached to the RapidIO network.

2.6 Subsystem Interfaces

RapidIO interconnect specification defines features that may be used to provide one or more common service layers for all participating RapidIO devices. These common services may act separately from device-specific drivers or be used by device-specific drivers. Example of such service provider is the RIONET driver which implements Ethernet-over-RapidIO interface. Because only one driver can be registered for a device, all common RapidIO services have to be registered as subsystem interfaces. This allows to have multiple common services attached to the same device without blocking attachment of a device-specific driver.

44.1.3 3. Subsystem Initialization

In order to initialize the RapidIO subsystem, a platform must initialize and register at least one master port within the RapidIO network. To register mport within the subsystem controller driver's initialization code calls function rio_register_mport() for each available master port.

After all active master ports are registered with a RapidIO subsystem, an enumeration and/or discovery routine may be called automatically or by user-space command.

RapidIO subsystem can be configured to be built as a statically linked or modular component of the kernel (see details below).

44.1.4 4. Enumeration and Discovery

4.1 Overview

RapidIO subsystem configuration options allow users to build enumeration and discovery methods as statically linked components or loadable modules. An enumeration/discovery method implementation and available input parameters define how any given method can be attached to available RapidIO mports: simply to all available mports OR individually to the specified mport device.

Depending on selected enumeration/discovery build configuration, there are several methods to initiate an enumeration and/or discovery process:

(a) Statically linked enumeration and discovery process can be started automatically during kernel initialization time using corresponding module parameters. This was the original method used since introduction of RapidIO subsystem. Now this method relies on enumerator module parameter which is 'rio-scan.scan' for existing basic enumeration/discovery method. When automatic start of enumeration/discovery is used a user has to ensure that all discovering endpoints are started before the enumerating endpoint and are waiting for enumeration to be completed. Configuration option CONFIG_RAPIDIO_DISC_TIMEOUT defines time that discovering endpoint waits for enumeration to be completed. If the specified timeout expires the discovery process is terminated without obtaining RapidIO network information. NOTE: a timed out discovery process may be restarted later using a user-space command as it is described below (if the given endpoint was enumerated successfully).

(b) Statically linked enumeration and discovery process can be started by a command from user space. This initiation method provides more flexibility for a system startup compared to the option (a) above. After all participating endpoints have been successfully booted, an enumeration process shall be started first by issuing a user-space command, after an enumeration is completed a discovery process can be started on all remaining endpoints.

(c) Modular enumeration and discovery process can be started by a command from user space. After an enumeration/discovery module is loaded, a network scan process can be started by issuing a user-space

command. Similar to the option (b) above, an enumerator has to be started first.

(d) Modular enumeration and discovery process can be started by a module initialization routine. In this case an enumerating module shall be loaded first.

When a network scan process is started it calls an enumeration or discovery routine depending on the configured role of a master port: host or agent.

Enumeration is performed by a master port if it is configured as a host port by assigning a host destination ID greater than or equal to zero. The host destination ID can be assigned to a master port using various methods depending on RapidIO subsystem build configuration:

(a) For a statically linked RapidIO subsystem core use command line parameter "rapidio.hdid=" with a list of destination ID assignments in order of mport device registration. For example, in a system with two RapidIO controllers the command line parameter "rapidio.hdid=-1,7"will result in assignment of the host destination ID=7 to the second RapidIO controller, while the first one will be assigned destination ID=-1.

(b) If the RapidIO subsystem core is built as a loadable module, in addition to the method shown above, the host destination ID(s) can be specified using traditional methods of passing module parameter "hdid=" during its loading:

- from command line: "modprobe rapidio hdid=-1,7" , or
- from modprobe configuration file using configuration command "options", like in this example: "options rapidio hdid=-1,7". An example of modprobe configuration file is provided in the section below.

NOTES: (i) if "hdid=" parameter is omitted all available mport will be assigned destination ID = -1;

(ii) the "hdid=" parameter in systems with multiple mports can have destination ID assignments omitted from the end of list (default = -1).

If the host device ID for a specific master port is set to -1, the discovery process will be performed for it.

The enumeration and discovery routines use RapidIO maintenance transactions to access the configuration space of devices.

NOTE: If RapidIO switch-specific device drivers are built as loadable modules they must be loaded before enumeration/discovery process starts. This requirement is cased by the fact that enumeration/discovery methods invoke vendor-specific callbacks on early stages.

4.2 Automatic Start of Enumeration and Discovery

Automatic enumeration/discovery start method is applicable only to built-in enumeration/discovery RapidIO configuration selection. To enable automatic enumeration/discovery start by existing basic enumerator method set use boot command line parameter "rio-scan.scan=1".

This configuration requires synchronized start of all RapidIO endpoints that form a network which will be enumerated/discovered. Discovering endpoints have to be started before an enumeration starts to ensure that all RapidIO controllers have been initialized and are ready to be discovered. Configuration parameter CONFIG_RAPIDIO_DISC_TIMEOUT defines time (in seconds) which a discovering endpoint will wait for enumeration to be completed.

When automatic enumeration/discovery start is selected, basic method's initialization routine calls rio_init_mports() to perform enumeration or discovery for all known mport devices.

Depending on RapidIO network size and configuration this automatic enumeration/discovery start method may be difficult to use due to the requirement for synchronized start of all endpoints.

4.3 User-space Start of Enumeration and Discovery

User-space start of enumeration and discovery can be used with built-in and modular build configurations. For user-space controlled start RapidIO subsystem creates the sysfs write-only attribute file '/sys/bus/rapidio/scan'. To initiate an enumeration or discovery process on specific mport device, a user needs to write mport_ID (not RapidIO destination ID) into that file. The mport_ID is a sequential number (0 ⋯RIO_MAX_MPORTS) assigned during mport device registration. For example for machine with single RapidIO controller, mport_ID for that controller always will be 0.

To initiate RapidIO enumeration/discovery on all available mports a user may write '-1' (or RIO_MPORT_ANY) into the scan attribute file.

4.4 Basic Enumeration Method

This is an original enumeration/discovery method which is available since first release of RapidIO subsystem code. The enumeration process is implemented according to the enumeration algorithm outlined in the RapidIO Interconnect Specification: Annex I [1].

This method can be configured as statically linked or loadable module. The method's single parameter "scan" allows to trigger the enumeration/discovery process from module initialization routine.

This enumeration/discovery method can be started only once and does not support unloading if it is built as a module.

The enumeration process traverses the network using a recursive depth-first algorithm. When a new device is found, the enumerator takes ownership of that device by writing into the Host Device ID Lock CSR. It does this to ensure that

the enumerator has exclusive right to enumerate the device. If device ownership is successfully acquired, the enumerator allocates a new rio_dev structure and initializes it according to device capabilities.

If the device is an endpoint, a unique device ID is assigned to it and its value is written into the device's Base Device ID CSR.

If the device is a switch, the enumerator allocates an additional rio_switch structure to store switch specific information. Then the switch's vendor ID and device ID are queried against a table of known RapidIO switches. Each switch table entry contains a pointer to a switch-specific initialization routine that initializes pointers to the rest of switch specific operations, and performs hardware initialization if necessary. A RapidIO switch does not have a unique device ID; it relies on hopcount and routing for device ID of an attached endpoint if access to its configuration registers is required. If a switch (or chain of switches) does not have any endpoint (except enumerator) attached to it, a fake device ID will be assigned to configure a route to that switch. In the case of a chain of switches without endpoint, one fake device ID is used to configure a route through the entire chain and switches are differentiated by their hopcount value.

For both endpoints and switches the enumerator writes a unique component tag into device's Component Tag CSR. That unique value is used by the error management notification mechanism to identify a device that is reporting an error management event.

Enumeration beyond a switch is completed by iterating over each active egress port of that switch. For each active link, a route to a default device ID (0xFF for 8-bit systems and 0xFFFF for 16-bit systems) is temporarily written into the routing table. The algorithm recurs by calling itself with hopcount + 1 and the default device ID in order to access the device on the active port.

After the host has completed enumeration of the entire network it releases devices by clearing device ID locks (calls rio_clear_locks()). For each endpoint in the system, it sets the Discovered bit in the Port General Control CSR to indicate that enumeration is completed and agents are allowed to execute passive discovery of the network.

The discovery process is performed by agents and is similar to the enumeration process that is described above. However, the discovery process is performed without changes to the existing routing because agents only gather information about RapidIO network structure and are building an internal map of discovered devices. This way each Linux-based component of the RapidIO subsystem has a complete view of the network. The discovery process can be performed simultaneously by several agents. After initializing its RapidIO master port each agent waits for enumeration completion by the host for the configured wait time period. If this wait time period expires before enumeration is completed, an agent skips RapidIO discovery and continues with remaining kernel initialization.

4.5 Adding New Enumeration/Discovery Method

RapidIO subsystem code organization allows addition of new enumeration/discovery methods as new configuration options without significant impact to the core RapidIO code.

A new enumeration/discovery method has to be attached to one or more mport devices before an enumeration/discovery process can be started. Normally, method's module initialization routine calls rio_register_scan() to attach an enumerator to a specified mport device (or devices). The basic enumerator implementation demonstrates this process.

4.6 Using Loadable RapidIO Switch Drivers

In the case when RapidIO switch drivers are built as loadable modules a user must ensure that they are loaded before the enumeration/discovery starts. This process can be automated by specifying pre- or post- dependencies in the RapidIO-specific modprobe configuration file as shown in the example below.

File /etc/modprobe.d/rapidio.conf:

```
# Configure RapidIO subsystem modules

# Set enumerator host destination ID (overrides kernel command line option)
options rapidio hdid=-1,2

# Load RapidIO switch drivers immediately after rapidio core module was␣
↪loaded
softdep rapidio post: idt_gen2 idtcps tsi57x

# OR :

# Load RapidIO switch drivers just before rio-scan enumerator module is␣
↪loaded
softdep rio-scan pre: idt_gen2 idtcps tsi57x

-------------------------
```

NOTE: In the example above, one of "softdep" commands must be removed or commented out to keep required module loading sequence.

44.1.5 5. References

[1] RapidIO Trade Association. RapidIO Interconnect Specifications. http://www.rapidio.org.

[2] Rapidio TA. Technology Comparisons. http://www.rapidio.org/education/technology_comparisons/

[3] RapidIO support for Linux. http://lwn.net/Articles/139118/

[4] Matt Porter. RapidIO for Linux. Ottawa Linux Symposium, 2005 http://www.kernel.org/doc/ols/2005/ols2005v2-pages-43-56.pdf

44.2 Sysfs entries

The RapidIO sysfs files have moved to: Documentation/ABI/testing/sysfs-bus-rapidio and Documentation/ABI/testing/sysfs-class-rapidio

44.3 RapidIO subsystem mport driver for IDT Tsi721 PCI Express-to-SRIO bridge.

44.3.1 1. Overview

This driver implements all currently defined RapidIO mport callback functions. It supports maintenance read and write operations, inbound and outbound RapidIO doorbells, inbound maintenance port-writes and RapidIO messaging.

To generate SRIO maintenance transactions this driver uses one of Tsi721 DMA channels. This mechanism provides access to larger range of hop counts and destination IDs without need for changes in outbound window translation.

RapidIO messaging support uses dedicated messaging channels for each mailbox. For inbound messages this driver uses destination ID matching to forward messages into the corresponding message queue. Messaging callbacks are implemented to be fully compatible with RIONET driver (Ethernet over RapidIO messaging services).

1. Module parameters:

- '**dbg_level**'
  - This parameter allows to control amount of debug information generated by this device driver. This parameter is formed by set of This parameter can be changed bit masks that correspond to the specific functional block. For mask definitions see 'drivers/rapidio/devices/tsi721.h' This parameter can be changed dynamically. Use CONFIG_RAPIDIO_DEBUG=y to enable debug output at the top level.
- '**dma_desc_per_channel**'
  - This parameter defines number of hardware buffer descriptors allocated for each registered Tsi721 DMA channel. Its default value is 128.
- '**dma_txqueue_sz**'
  - DMA transactions queue size. Defines number of pending transaction requests that can be accepted by each DMA channel. Default value is 16.
- '**dma_sel**'
  - DMA channel selection mask. Bitmask that defines which hardware DMA channels (0 ···6) will be registered with DmaEngine core. If bit is set to 1, the corresponding DMA channel will be registered. DMA

channels not selected by this mask will not be used by this device driver. Default value is 0x7f (use all channels).

- ‘**pcie_mrrs**’
  - override value for PCIe Maximum Read Request Size (MRRS). This parameter gives an ability to override MRRS value set during PCIe configuration process. Tsi721 supports read request sizes up to 4096B. Value for this parameter must be set as defined by PCIe specification: 0 = 128B, 1 = 256B, 2 = 512B, 3 = 1024B, 4 = 2048B and 5 = 4096B. Default value is ‘-1’ (= keep platform setting).
- ‘**mbox_sel**’
  - RIO messaging MBOX selection mask. This is a bitmask that defines messaging MBOXes are managed by this device driver. Mask bits 0 - 3 correspond to MBOX0 - MBOX3. MBOX is under driver’s control if the corresponding bit is set to ‘1’. Default value is 0x0f (= all).

44.3.2 2. Known problems

None.

44.3.3 3. DMA Engine Support

Tsi721 mport driver supports DMA data transfers between local system memory and remote RapidIO devices. This functionality is implemented according to SLAVE mode API defined by common Linux kernel DMA Engine framework.

Depending on system requirements RapidIO DMA operations can be included/excluded by setting CONFIG_RAPIDIO_DMA_ENGINE option. Tsi721 miniport driver uses seven out of eight available BDMA channels to support DMA data transfers. One BDMA channel is reserved for generation of maintenance read/write requests.

If Tsi721 mport driver have been built with RAPIDIO_DMA_ENGINE support included, this driver will accept DMA-specific module parameter:

“dma_desc_per_channel”

- defines number of hardware buffer descriptors used by each BDMA channel of Tsi721 (by default - 128).

4. Version History

1.1.0	DMA operations re-worked to support data scatter/gather lists larger than hardware buffer descriptors ring.
1.0.0	Initial driver release.

44.3.4 5. License

44.4 RapidIO subsystem mport character device driver (rio_mport_cdev.c)

44.4.1 1. Overview

This device driver is the result of collaboration within the RapidIO.org Software Task Group (STG) between Texas Instruments, Freescale, Prodrive Technologies, Nokia Networks, BAE and IDT. Additional input was received from other members of RapidIO.org. The objective was to create a character mode driver interface which exposes the capabilities of RapidIO devices directly to applications, in a manner that allows the numerous and varied RapidIO implementations to interoperate.

This driver (MPORT_CDEV) provides access to basic RapidIO subsystem operations for user-space applications. Most of RapidIO operations are supported through 'ioctl' system calls.

When loaded this device driver creates filesystem nodes named rio_mportX in /dev directory for each registered RapidIO mport device. 'X' in the node name matches to unique port ID assigned to each local mport device.

Using available set of ioctl commands user-space applications can perform following RapidIO bus and subsystem operations:

- Reads and writes from/to configuration registers of mport devices (RIO_MPORT_MAINT_READ_LOCAL/RIO_MPORT_MAINT_WRITE_LOCAL)
- Reads and writes from/to configuration registers of remote RapidIO devices. This operations are defined as RapidIO Maintenance reads/writes in RIO spec. (RIO_MPORT_MAINT_READ_REMOTE/RIO_MPORT_MAINT_WRITE_REMOTE)
- Set RapidIO Destination ID for mport devices (RIO_MPORT_MAINT_HDID_SET)

- Set RapidIO Component Tag for mport devices (RIO_MPORT_MAINT_COMPTAG_SET)
- Query logical index of mport devices (RIO_MPORT_MAINT_PORT_IDX_GET)
- Query capabilities and RapidIO link configuration of mport devices (RIO_MPORT_GET_PROPERTIES)
- Enable/Disable reporting of RapidIO doorbell events to user-space applications (RIO_ENABLE_DOORBELL_RANGE/RIO_DISABLE_DOORBELL_RANGE)
- Enable/Disable reporting of RIO port-write events to user-space applications (RIO_ENABLE_PORTWRITE_RANGE/RIO_DISABLE_PORTWRITE_RANGE)
- Query/Control type of events reported through this driver: doorbells, port-writes or both (RIO_SET_EVENT_MASK/RIO_GET_EVENT_MASK)
- Configure/Map mport's outbound requests window(s) for specific size, RapidIO destination ID, hopcount and request type (RIO_MAP_OUTBOUND/RIO_UNMAP_OUTBOUND)
- Configure/Map mport's inbound requests window(s) for specific size, RapidIO base address and local memory base address (RIO_MAP_INBOUND/RIO_UNMAP_INBOUND)
- Allocate/Free contiguous DMA coherent memory buffer for DMA data transfers to/from remote RapidIO devices (RIO_ALLOC_DMA/RIO_FREE_DMA)
- Initiate DMA data transfers to/from remote RapidIO devices (RIO_TRANSFER). Supports blocking, asynchronous and posted (a.k.a 'fire-and-forget') data transfer modes.
- Check/Wait for completion of asynchronous DMA data transfer (RIO_WAIT_FOR_ASYNC)
- Manage device objects supported by RapidIO subsystem (RIO_DEV_ADD/RIO_DEV_DEL). This allows implementation of various RapidIO fabric enumeration algorithms as user-space applications while using remaining functionality provided by kernel RapidIO subsystem.

44.4.2 2. Hardware Compatibility

This device driver uses standard interfaces defined by kernel RapidIO subsystem and therefore it can be used with any mport device driver registered by RapidIO subsystem with limitations set by available mport implementation.

At this moment the most common limitation is availability of RapidIO-specific DMA engine framework for specific mport device. Users should verify available functionality of their platform when planning to use this driver:

- IDT Tsi721 PCIe-to-RapidIO bridge device and its mport device driver are fully compatible with this driver.
- Freescale SoCs 'fsl_rio' mport driver does not have implementation for RapidIO specific DMA engine support and therefore DMA data transfers mport_cdev driver are not available.

44.4.3 3. Module parameters

- **'dma_timeout'**
  - DMA transfer completion timeout (in msec, default value 3000). This parameter set a maximum completion wait time for SYNC mode DMA transfer requests and for RIO_WAIT_FOR_ASYNC ioctl requests.
- **'dbg_level'**
  - This parameter allows to control amount of debug information generated by this device driver. This parameter is formed by set of bit masks that correspond to the specific functional blocks. For mask definitions see 'drivers/rapidio/devices/rio_mport_cdev.c' This parameter can be changed dynamically. Use CONFIG_RAPIDIO_DEBUG=y to enable debug output at the top level.

44.4.4 4. Known problems

None.

44.4.5 5. User-space Applications and API

API library and applications that use this device driver are available from RapidIO.org.

44.4.6 6. TODO List

- Add support for sending/receiving "raw" RapidIO messaging packets.
- Add memory mapped DMA data transfers as an option when RapidIO-specific DMA is not available.

44.5 RapidIO subsystem Channelized Messaging character device driver (rio_cm.c)

44.5.1 1. Overview

This device driver is the result of collaboration within the RapidIO.org Software Task Group (STG) between Texas Instruments, Prodrive Technologies, Nokia Networks, BAE and IDT. Additional input was received from other members of RapidIO.org.

The objective was to create a character mode driver interface which exposes messaging capabilities of RapidIO endpoint devices (mports) directly to applications, in a manner that allows the numerous and varied RapidIO implementations to interoperate.

This driver (RIO_CM) provides to user-space applications shared access to RapidIO mailbox messaging resources.

RapidIO specification (Part 2) defines that endpoint devices may have up to four messaging mailboxes in case of multi-packet message (up to 4KB) and up to 64 mailboxes if single-packet messages (up to 256 B) are used. In addition to protocol definition limitations, a particular hardware implementation can have reduced number of messaging mailboxes. RapidIO aware applications must therefore share the messaging resources of a RapidIO endpoint.

Main purpose of this device driver is to provide RapidIO mailbox messaging capability to large number of user-space processes by introducing socket-like operations using a single messaging mailbox. This allows applications to use the limited RapidIO messaging hardware resources efficiently.

Most of device driver' s operations are supported through 'ioctl' system calls.

When loaded this device driver creates a single file system node named rio_cm in /dev directory common for all registered RapidIO mport devices.

Following ioctl commands are available to user-space applications:

- **RIO_CM_MPORT_GET_LIST:** Returns to caller list of local mport devices that support messaging operations (number of entries up to RIO_MAX_MPORTS). Each list entry is combination of mport' s index in the system and RapidIO destination ID assigned to the port.
- **RIO_CM_EP_GET_LIST_SIZE:** Returns number of messaging capable remote endpoints in a RapidIO network associated with the specified mport device.
- **RIO_CM_EP_GET_LIST:** Returns list of RapidIO destination IDs for messaging capable remote endpoints (peers) available in a RapidIO network associated with the specified mport device.
- **RIO_CM_CHAN_CREATE:** Creates RapidIO message exchange channel data structure with channel ID assigned automatically or as requested by a caller.
- **RIO_CM_CHAN_BIND:** Binds the specified channel data structure to the specified mport device.
- **RIO_CM_CHAN_LISTEN:** Enables listening for connection requests on the specified channel.
- **RIO_CM_CHAN_ACCEPT:** Accepts a connection request from peer on the specified channel. If wait timeout for this request is specified by a caller it is a blocking call. If timeout set to 0 this is non-blocking call - ioctl handler checks for a pending connection request and if one is not available exits with -EGAIN error status immediately.
- **RIO_CM_CHAN_CONNECT:** Sends a connection request to a remote peer/channel.
- **RIO_CM_CHAN_SEND:** Sends a data message through the specified channel. The handler for this request assumes that message buffer specified by a caller includes the reserved space for a packet header required by this driver.
- **RIO_CM_CHAN_RECEIVE:** Receives a data message through a connected channel. If the channel does not have an incoming message ready to

return this ioctl handler will wait for new message until timeout specified by a caller expires. If timeout value is set to 0, ioctl handler uses a default value defined by MAX_SCHEDULE_TIMEOUT.

- **RIO_CM_CHAN_CLOSE:** Closes a specified channel and frees associated buffers. If the specified channel is in the CONNECTED state, sends close notification to the remote peer.

The ioctl command codes and corresponding data structures intended for use by user-space applications are defined in 'include/uapi/linux/rio_cm_cdev.h'.

44.5.2 2. Hardware Compatibility

This device driver uses standard interfaces defined by kernel RapidIO subsystem and therefore it can be used with any mport device driver registered by RapidIO subsystem with limitations set by available mport HW implementation of messaging mailboxes.

44.5.3 3. Module parameters

- '**dbg_level**'
  - This parameter allows to control amount of debug information generated by this device driver. This parameter is formed by set of bit masks that correspond to the specific functional block. For mask definitions see 'drivers/rapidio/devices/rio_cm.c' This parameter can be changed dynamically. Use CONFIG_RAPIDIO_DEBUG=y to enable debug output at the top level.
- '**cmbox**'
  - Number of RapidIO mailbox to use (default value is 1). This parameter allows to set messaging mailbox number that will be used within entire RapidIO network. It can be used when default mailbox is used by other device drivers or is not supported by some nodes in the RapidIO network.
- '**chstart**'
  - Start channel number for dynamic assignment. Default value - 256. Allows to exclude channel numbers below this parameter from dynamic allocation to avoid conflicts with software components that use reserved predefined channel numbers.

44.5.4 4. Known problems

None.

44.5.5 5. User-space Applications and API Library

Messaging API library and applications that use this device driver are available from RapidIO.org.

44.5.6 6. TODO List

- Add support for system notification messages (reserved channel 0).

CHAPTER

FORTYFIVE

WRITING S390 CHANNEL DEVICE DRIVERS

Author Cornelia Huck

45.1 Introduction

This document describes the interfaces available for device drivers that drive s390 based channel attached I/O devices. This includes interfaces for interaction with the hardware and interfaces for interacting with the common driver core. Those interfaces are provided by the s390 common I/O layer.

The document assumes a familarity with the technical terms associated with the s390 channel I/O architecture. For a description of this architecture, please refer to the "z/Architecture: Principles of Operation", IBM publication no. SA22-7832.

While most I/O devices on a s390 system are typically driven through the channel I/O mechanism described here, there are various other methods (like the diag interface). These are out of the scope of this document.

The s390 common I/O layer also provides access to some devices that are not strictly considered I/O devices. They are considered here as well, although they are not the focus of this document.

Some additional information can also be found in the kernel source under Documentation/s390/driver-model.rst.

45.2 The css bus

The css bus contains the subchannels available on the system. They fall into several categories:

- Standard I/O subchannels, for use by the system. They have a child device on the ccw bus and are described below.
- I/O subchannels bound to the vfio-ccw driver. See Documentation/s390/vfio-ccw.rst.
- Message subchannels. No Linux driver currently exists.
- CHSC subchannels (at most one). The chsc subchannel driver can be used to send asynchronous chsc commands.
- eADM subchannels. Used for talking to storage class memory.

45.3 The ccw bus

The ccw bus typically contains the majority of devices available to a s390 system. Named after the channel command word (ccw), the basic command structure used to address its devices, the ccw bus contains so-called channel attached devices. They are addressed via I/O subchannels, visible on the css bus. A device driver for channel-attached devices, however, will never interact with the subchannel directly, but only via the I/O device on the ccw bus, the ccw device.

45.3.1 I/O functions for channel-attached devices

Some hardware structures have been translated into C structures for use by the common I/O layer and device drivers. For more information on the hardware structures represented here, please consult the Principles of Operation.

struct **ccw1**
channel command word

Definition

```
struct ccw1 {
  __u8 cmd_code;
  __u8 flags;
  __u16 count;
  __u32 cda;
};
```

Members

cmd_code command code

flags flags, like IDA addressing, etc.

count byte count

cda data address

Description

The ccw is the basic structure to build channel programs that perform operations with the device or the control unit. Only Format-1 channel command words are supported.

struct **ccw0**
channel command word

Definition

```
struct ccw0 {
  __u8 cmd_code;
  __u32 cda : 24;
  __u8 flags;
  __u8 reserved;
  __u16 count;
};
```

Members

cmd_code command code

cda data address

flags flags, like IDA addressing, etc.

reserved will be ignored

count byte count

Description

The format-0 ccw structure.

struct **erw**
extended report word

Definition

```
struct erw {
  __u32 res0  : 3;
  __u32 auth  : 1;
  __u32 pvrf  : 1;
  __u32 cpt   : 1;
  __u32 fsavf : 1;
  __u32 cons  : 1;
  __u32 scavf : 1;
  __u32 fsaf  : 1;
  __u32 scnt  : 6;
  __u32 res16 : 16;
};
```

Members

res0 reserved

auth authorization check

pvrf path-verification-required flag

cpt channel-path timeout

fsavf failing storage address validity flag

cons concurrent sense

scavf secondary ccw address validity flag

fsaf failing storage address format

scnt sense count, if **cons** == 1

res16 reserved

struct **erw_eadm**
EADM Subchannel extended report word

Definition

```
struct erw_eadm {
  __u32 : 16;
  __u32 b : 1;
  __u32 r : 1;
```

(continues on next page)

(continued from previous page)

```
  __u32 : 14;
};
```

Members

b aob error

r arsb error

struct **sublog**
subchannel logout area

Definition

```
struct sublog {
  __u32 res0  : 1;
  __u32 esf   : 7;
  __u32 lpum  : 8;
  __u32 arep  : 1;
  __u32 fvf   : 5;
  __u32 sacc  : 2;
  __u32 termc : 2;
  __u32 devsc : 1;
  __u32 serr  : 1;
  __u32 ioerr : 1;
  __u32 seqc  : 3;
};
```

Members

res0 reserved

esf extended status flags

lpum last path used mask

arep ancillary report

fvf field-validity flags

sacc storage access code

termc termination code

devsc device-status check

serr secondary error

ioerr i/o-error alert

seqc sequence code

struct **esw0**
Format 0 Extended Status Word (ESW)

Definition

```
struct esw0 {
  struct sublog sublog;
  struct erw erw;
```

(continues on next page)

(continued from previous page)

```
  __u32 faddr[2];
  __u32 saddr;
};
```

Members

sublog subchannel logout

erw extended report word

faddr failing storage address

saddr secondary ccw address

struct **esw1**
Format 1 Extended Status Word (ESW)

Definition

```
struct esw1 {
  __u8 zero0;
  __u8 lpum;
  __u16 zero16;
  struct erw erw;
  __u32 zeros[3];
};
```

Members

zero0 reserved zeros

lpum last path used mask

zero16 reserved zeros

erw extended report word

zeros three fullwords of zeros

struct **esw2**
Format 2 Extended Status Word (ESW)

Definition

```
struct esw2 {
  __u8 zero0;
  __u8 lpum;
  __u16 dcti;
  struct erw erw;
  __u32 zeros[3];
};
```

Members

zero0 reserved zeros

lpum last path used mask

dcti device-connect-time interval

erw extended report word

zeros three fullwords of zeros

struct **esw3**
Format 3 Extended Status Word (ESW)

Definition

```
struct esw3 {
  __u8 zero0;
  __u8 lpum;
  __u16 res;
  struct erw erw;
  __u32 zeros[3];
};
```

Members

zero0 reserved zeros

lpum last path used mask

res reserved

erw extended report word

zeros three fullwords of zeros

struct **esw_eadm**
EADM Subchannel Extended Status Word (ESW)

Definition

```
struct esw_eadm {
  __u32 sublog;
  struct erw_eadm erw;
  __u32 : 32;
  __u32 : 32;
  __u32 : 32;
};
```

Members

sublog subchannel logout

erw extended report word

struct **irb**
interruption response block

Definition

```
struct irb {
  union scsw scsw;
  union {
    struct esw0 esw0;
    struct esw1 esw1;
    struct esw2 esw2;
    struct esw3 esw3;
    struct esw_eadm eadm;
  } esw;
```

(continues on next page)

(continued from previous page)

```
  __u8 ecw[32];
};
```

Members

scsw subchannel status word

esw extended status word

ecw extended control word

Description

The irb that is handed to the device driver when an interrupt occurs. For solicited interrupts, the common I/O layer already performs checks whether a field is valid; a field not being valid is always passed as `0`. If a unit check occurred, **ecw** may contain sense data; this is retrieved by the common I/O layer itself if the device doesn't support concurrent sense (so that the device driver never needs to perform basic sense itself). For unsolicited interrupts, the irb is passed as-is (expect for sense data, if applicable).

struct **ciw**
command information word (CIW) layout

Definition

```
struct ciw {
  __u32 et       :  2;
  __u32 reserved :  2;
  __u32 ct       :  4;
  __u32 cmd      :  8;
  __u32 count    : 16;
};
```

Members

et entry type

reserved reserved bits

ct command type

cmd command code

count command count

struct **ccw_dev_id**
unique identifier for ccw devices

Definition

```
struct ccw_dev_id {
  u8 ssid;
  u16 devno;
};
```

Members

ssid subchannel set id

devno device number

Description

This structure is not directly based on any hardware structure. The hardware identifies a device by its device number and its subchannel, which is in turn identified by its id. In order to get a unique identifier for ccw devices across subchannel sets, **struct** ccw_dev_id has been introduced.

int **ccw_dev_id_is_equal**(struct ccw_dev_id * dev_id1, struct ccw_dev_id * dev_id2)
compare two ccw_dev_ids

Parameters

struct ccw_dev_id * dev_id1 a ccw_dev_id

struct ccw_dev_id * dev_id2 another ccw_dev_id

Return

`1` if the two structures are equal field-by-field, `0` if not.

Context

any

u8 **pathmask_to_pos**(u8 mask)
find the position of the left-most bit in a pathmask

Parameters

u8 mask pathmask with at least one bit set

45.3.2 ccw devices

Devices that want to initiate channel I/O need to attach to the ccw bus. Interaction with the driver core is done via the common I/O layer, which provides the abstractions of ccw devices and ccw device drivers.

The functions that initiate or terminate channel I/O all act upon a ccw device structure. Device drivers must not bypass those functions or strange side effects may happen.

struct **ccw_device**
channel attached device

Definition

```
struct ccw_device {
  spinlock_t *ccwlock;
  struct ccw_device_id id;
  struct ccw_driver *drv;
  struct device dev;
  int online;
  void (*handler) (struct ccw_device *, unsigned long, struct irb *);
};
```

Members

ccwlock pointer to device lock

id id of this device

drv ccw driver for this device

dev embedded device structure

online online status of device

handler interrupt handler

Description

handler is a member of the device rather than the driver since a driver can have different interrupt handlers for different ccw devices (multi-subchannel drivers).

struct **ccw_driver**
device driver for channel attached devices

Definition

```
struct ccw_driver {
  struct ccw_device_id *ids;
  int (*probe) (struct ccw_device *);
  void (*remove) (struct ccw_device *);
  int (*set_online) (struct ccw_device *);
  int (*set_offline) (struct ccw_device *);
  int (*notify) (struct ccw_device *, int);
  void (*path_event) (struct ccw_device *, int *);
  void (*shutdown) (struct ccw_device *);
  int (*prepare) (struct ccw_device *);
  void (*complete) (struct ccw_device *);
  int (*freeze)(struct ccw_device *);
  int (*thaw) (struct ccw_device *);
  int (*restore)(struct ccw_device *);
  enum uc_todo (*uc_handler) (struct ccw_device *, struct irb *);
  struct device_driver driver;
  enum interruption_class int_class;
};
```

Members

ids ids supported by this driver

probe function called on probe

remove function called on remove

set_online called when setting device online

set_offline called when setting device offline

notify notify driver of device state changes

path_event notify driver of channel path events

shutdown called at device shutdown

prepare prepare for pm state transition

complete undo work done in **prepare**

freeze callback for freezing during hibernation snapshotting

thaw undo work done in **freeze**

restore callback for restoring after hibernation

uc_handler callback for unit check handler

driver embedded device driver structure

int_class interruption class to use for accounting interrupts

int **ccw_device_set_offline**(struct ccw_device * cdev)
disable a ccw device for I/O

Parameters

struct ccw_device * cdev target ccw device

Description

This function calls the driver's set_offline() function for **cdev**, if given, and then disables **cdev**.

Return

`0` on success and a negative error value on failure.

Context

enabled, ccw device lock not held

int **ccw_device_set_online**(struct ccw_device * cdev)
enable a ccw device for I/O

Parameters

struct ccw_device * cdev target ccw device

Description

This function first enables **cdev** and then calls the driver's set_online() function for **cdev**, if given. If set_online() returns an error, **cdev** is disabled again.

Return

`0` on success and a negative error value on failure.

Context

enabled, ccw device lock not held

struct ccw_device * **get_ccwdev_by_dev_id**(struct ccw_dev_id * dev_id)
obtain device from a ccw device id

Parameters

struct ccw_dev_id * dev_id id of the device to be searched

Description

This function searches all devices attached to the ccw bus for a device matching **dev_id**.

Return

If a device is found its reference count is increased and returned; else `NULL` is returned.

struct ccw_device * **get_ccwdev_by_busid**(struct ccw_driver * cdrv, const char * bus_id)
obtain device from a bus id

Parameters

struct ccw_driver * cdrv driver the device is owned by

const char * bus_id bus id of the device to be searched

Description

This function searches all devices owned by **cdrv** for a device with a bus id matching **bus_id**.

Return

If a match is found, its reference count of the found device is increased and it is returned; else `NULL` is returned.

int **ccw_driver_register**(struct ccw_driver * cdriver)
register a ccw driver

Parameters

struct ccw_driver * cdriver driver to be registered

Description

This function is mainly a wrapper around `driver_register()`.

Return

`0` on success and a negative error value on failure.

void **ccw_driver_unregister**(struct ccw_driver * cdriver)
deregister a ccw driver

Parameters

struct ccw_driver * cdriver driver to be deregistered

Description

This function is mainly a wrapper around `driver_unregister()`.

int **ccw_device_siosl**(struct ccw_device * cdev)
initiate logging

Parameters

struct ccw_device * cdev ccw device

Description

This function is used to invoke model-dependent logging within the channel subsystem.

int **ccw_device_set_options_mask**(struct ccw_device * cdev, unsigned long flags)
set some options and unset the rest

Parameters

struct ccw_device * cdev device for which the options are to be set

unsigned long flags options to be set

Description

All flags specified in **flags** are set, all flags not specified in **flags** are cleared.

Return

`0` on success, `-EINVAL` on an invalid flag combination.

int **ccw_device_set_options**(struct ccw_device * cdev, unsigned long flags)
set some options

Parameters

struct ccw_device * cdev device for which the options are to be set

unsigned long flags options to be set

Description

All flags specified in **flags** are set, the remainder is left untouched.

Return

`0` on success, `-EINVAL` if an invalid flag combination would ensue.

void **ccw_device_clear_options**(struct ccw_device * cdev, unsigned long flags)
clear some options

Parameters

struct ccw_device * cdev device for which the options are to be cleared

unsigned long flags options to be cleared

Description

All flags specified in **flags** are cleared, the remainder is left untouched.

int **ccw_device_is_pathgroup**(struct ccw_device * cdev)
determine if paths to this device are grouped

Parameters

struct ccw_device * cdev ccw device

Description

Return non-zero if there is a path group, zero otherwise.

int **ccw_device_is_multipath**(struct ccw_device * cdev)
determine if device is operating in multipath mode

Parameters

struct ccw_device * cdev ccw device

Description

Return non-zero if device is operating in multipath mode, zero otherwise.

int **ccw_device_clear**(struct ccw_device * cdev, unsigned long intparm)
terminate I/O request processing

Parameters

struct ccw_device * cdev target ccw device

unsigned long intparm interruption parameter to be returned upon conclusion of csch

Description

ccw_device_clear() calls csch on **cdev**'s subchannel.

Return

0 on success, -ENODEV on device not operational, -EINVAL on invalid device state.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_start_timeout_key**(struct ccw_device * cdev, struct ccw1 * cpa, unsigned long intparm, __u8 lpm, __u8 key, unsigned long flags, int expires)

start a s390 channel program with timeout and key

Parameters

struct ccw_device * cdev target ccw device

struct ccw1 * cpa logical start address of channel program

unsigned long intparm user specific interruption parameter; will be presented back to **cdev**'s interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.

__u8 lpm defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.

__u8 key storage key to be used for the I/O

unsigned long flags additional flags; defines the action to be performed for I/O processing.

int expires timeout value in jiffies

Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered). This function notifies the device driver if the channel program has not completed during the time specified by **expires**. If a timeout occurs, the channel program is terminated via xsch, hsch or csch, and the device's interrupt handler will be called with an irb containing ERR_PTR(-ETIMEDOUT). The interruption handler will echo back the **intparm** specified here, unless another interruption parameter is specified by a subsequent invocation of ccw_device_halt() or ccw_device_clear().

Return

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in **lpm** is operational; -ENODEV, if the device is not operational.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_start_key**(struct ccw_device * cdev, struct ccw1 * cpa, unsigned long intparm, __u8 lpm, __u8 key, unsigned long flags)
start a s390 channel program with key

Parameters

struct ccw_device * cdev target ccw device

struct ccw1 * cpa logical start address of channel program

unsigned long intparm user specific interruption parameter; will be presented back to **cdev**'s interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.

__u8 lpm defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.

__u8 key storage key to be used for the I/O

unsigned long flags additional flags; defines the action to be performed for I/O processing.

Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered). The interruption handler will echo back the **intparm** specified here, unless another interruption parameter is specified by a subsequent invocation of ccw_device_halt() or ccw_device_clear().

Return

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in **lpm** is operational; -ENODEV, if the device is not operational.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_start**(struct ccw_device * cdev, struct ccw1 * cpa, unsigned long intparm, __u8 lpm, unsigned long flags)
start a s390 channel program

Parameters

struct ccw_device * cdev target ccw device

struct ccw1 * cpa logical start address of channel program

unsigned long intparm user specific interruption parameter; will be presented back to **cdev**'s interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.

__u8 lpm defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.

unsigned long flags additional flags; defines the action to be performed for I/O processing.

Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered). The interruption handler will echo back the **intparm** specified here, unless another interruption parameter is specified by a subsequent invocation of `ccw_device_halt()` or `ccw_device_clear()`.

Return

`0`, if the operation was successful; `-EBUSY`, if the device is busy, or status pending; `-EACCES`, if no path specified in **lpm** is operational; `-ENODEV`, if the device is not operational.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_start_timeout**(struct ccw_device * cdev, struct ccw1 * cpa, unsigned long intparm, __u8 lpm, unsigned long flags, int expires)
start a s390 channel program with timeout

Parameters

struct ccw_device * cdev target ccw device

struct ccw1 * cpa logical start address of channel program

unsigned long intparm user specific interruption parameter; will be presented back to **cdev**'s interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.

__u8 lpm defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.

unsigned long flags additional flags; defines the action to be performed for I/O processing.

int expires timeout value in jiffies

Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered). This function notifies the device driver if the channel program has not completed during the time specified by **expires**. If a timeout occurs, the channel program is terminated via xsch, hsch or csch, and the device's interrupt handler will be called with an irb containing ERR_PTR(`-ETIMEDOUT`). The interruption handler will echo back the **intparm** specified here, unless another interruption parameter is specified by a subsequent invocation of `ccw_device_halt()` or `ccw_device_clear()`.

Return

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in **lpm** is operational; -ENODEV, if the device is not operational.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_halt**(struct ccw_device * cdev, unsigned long intparm)
halt I/O request processing

Parameters

struct ccw_device * cdev target ccw device

unsigned long intparm interruption parameter to be returned upon conclusion of hsch

Description

ccw_device_halt() calls hsch on **cdev**'s subchannel. The interruption handler will echo back the **intparm** specified here, unless another interruption parameter is specified by a subsequent invocation of ccw_device_clear().

Return

0 on success, -ENODEV on device not operational, -EINVAL on invalid device state, -EBUSY on device busy or interrupt pending.

Context

Interrupts disabled, ccw device lock held

int **ccw_device_resume**(struct ccw_device * cdev)
resume channel program execution

Parameters

struct ccw_device * cdev target ccw device

Description

ccw_device_resume() calls rsch on **cdev**'s subchannel.

Return

0 on success, -ENODEV on device not operational, -EINVAL on invalid device state, -EBUSY on device busy or interrupt pending.

Context

Interrupts disabled, ccw device lock held

struct ciw * **ccw_device_get_ciw**(struct ccw_device * cdev, __u32 ct)
Search for CIW command in extended sense data.

Parameters

struct ccw_device * cdev ccw device to inspect

__u32 ct command type to look for

Description

During SenseID, command information words (CIWs) describing special commands available to the device may have been stored in the extended sense data. This function searches for CIWs of a specified command type in the extended sense data.

Return

`NULL` if no extended sense data has been stored or if no CIW of the specified command type could be found, else a pointer to the CIW of the specified command type.

__u8 **ccw_device_get_path_mask**(struct ccw_device * cdev)
get currently available paths

Parameters

struct ccw_device * cdev ccw device to be queried

Return

`0` if no subchannel for the device is available, else the mask of currently available paths for the ccw device's subchannel.

struct channel_path_desc_fmt0 * **ccw_device_get_chp_desc**(struct ccw_device * cdev, int chp_idx)
return newly allocated channel-path descriptor

Parameters

struct ccw_device * cdev device to obtain the descriptor for

int chp_idx index of the channel path

Description

On success return a newly allocated copy of the channel-path description data associated with the given channel path. Return `NULL` on error.

u8 * **ccw_device_get_util_str**(struct ccw_device * cdev, int chp_idx)
return newly allocated utility strings

Parameters

struct ccw_device * cdev device to obtain the utility strings for

int chp_idx index of the channel path

Description

On success return a newly allocated copy of the utility strings associated with the given channel path. Return `NULL` on error.

void **ccw_device_get_id**(struct ccw_device * cdev, struct ccw_dev_id * dev_id)
obtain a ccw device id

Parameters

struct ccw_device * cdev device to obtain the id for

struct ccw_dev_id * dev_id where to fill in the values

int **ccw_device_tm_start_timeout_key**(struct ccw_device * cdev, struct tcw * tcw, unsigned long intparm, u8 lpm, u8 key, int expires)
perform start function

Parameters

struct ccw_device * cdev ccw device on which to perform the start function

struct tcw * tcw transport-command word to be started

unsigned long intparm user defined parameter to be passed to the interrupt handler

u8 lpm mask of paths to use

u8 key storage key to use for storage access

int expires time span in jiffies after which to abort request

Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

int **ccw_device_tm_start_key**(struct ccw_device * cdev, struct tcw * tcw, unsigned long intparm, u8 lpm, u8 key)
perform start function

Parameters

struct ccw_device * cdev ccw device on which to perform the start function

struct tcw * tcw transport-command word to be started

unsigned long intparm user defined parameter to be passed to the interrupt handler

u8 lpm mask of paths to use

u8 key storage key to use for storage access

Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

int **ccw_device_tm_start**(struct ccw_device * cdev, struct tcw * tcw, unsigned long intparm, u8 lpm)
perform start function

Parameters

struct ccw_device * cdev ccw device on which to perform the start function

struct tcw * tcw transport-command word to be started

unsigned long intparm user defined parameter to be passed to the interrupt handler

u8 lpm mask of paths to use

Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

int **ccw_device_tm_start_timeout**(struct ccw_device * cdev, struct tcw * tcw, unsigned long intparm, u8 lpm, int expires)
perform start function

Parameters

struct ccw_device * cdev ccw device on which to perform the start function

struct tcw * tcw transport-command word to be started

unsigned long intparm user defined parameter to be passed to the interrupt handler

u8 lpm mask of paths to use

int expires time span in jiffies after which to abort request

Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

int **ccw_device_get_mdc**(struct ccw_device * cdev, u8 mask)
accumulate max data count

Parameters

struct ccw_device * cdev ccw device for which the max data count is accumulated

u8 mask mask of paths to use

Description

Return the number of 64K-bytes blocks all paths at least support for a transport command. Return value 0 indicates failure.

int **ccw_device_tm_intrg**(struct ccw_device * cdev)
perform interrogate function

Parameters

struct ccw_device * cdev ccw device on which to perform the interrogate function

Description

Perform an interrogate function on the given ccw device. Return zero on success, non-zero otherwise.

void **ccw_device_get_schid**(struct ccw_device * cdev, struct subchannel_id * schid)
obtain a subchannel id

Parameters

struct ccw_device * cdev device to obtain the id for

struct subchannel_id * schid where to fill in the values

int **ccw_device_pnso**(struct ccw_device * cdev, struct chsc_pnso_area * pnso_area, struct chsc_pnso_resume_token resume_token, int cnc)
Perform Network-Subchannel Operation

Parameters

struct ccw_device * cdev device on which PNSO is performed

struct chsc_pnso_area * pnso_area request and response block for the operation

struct chsc_pnso_resume_token resume_token resume token for multiblock response

int cnc Boolean change-notification control

Description

pnso_area must be allocated by the caller with get_zeroed_page(GFP_KERNEL)

Returns 0 on success.

45.3.3 The channel-measurement facility

The channel-measurement facility provides a means to collect measurement data which is made available by the channel subsystem for each channel attached device.

struct **cmbdata**
channel measurement block data for user space

Definition

```
struct cmbdata {
  __u64 size;
  __u64 elapsed_time;
  __u64 ssch_rsch_count;
  __u64 sample_count;
  __u64 device_connect_time;
  __u64 function_pending_time;
  __u64 device_disconnect_time;
  __u64 control_unit_queuing_time;
  __u64 device_active_only_time;
  __u64 device_busy_time;
  __u64 initial_command_response_time;
};
```

Members

size size of the stored data

elapsed_time time since last sampling

ssch_rsch_count number of ssch and rsch

sample_count number of samples

device_connect_time time of device connect

function_pending_time time of function pending

device_disconnect_time time of device disconnect

control_unit_queuing_time time of control unit queuing

device_active_only_time time of device active only

device_busy_time time of device busy (ext. format)

initial_command_response_time initial command response time (ext. format)

Description

All values are stored as 64 bit for simplicity, especially in 32 bit emulation mode. All time values are normalized to nanoseconds. Currently, two formats are known, which differ by the size of this structure, i.e. the last two members are only set when the extended channel measurement facility (first shipped in z990 machines) is activated. Potentially, more fields could be added, which would result in a new ioctl number.

int **enable_cmf**(struct ccw_device * cdev)
switch on the channel measurement for a specific device

Parameters

struct ccw_device * cdev The ccw device to be enabled

Enable channel measurements for **cdev**. If this is called on a device for which channel measurement is already enabled a reset of the measurement data is triggered.

Return

0 for success or a negative error value.

Context

non-atomic

int **disable_cmf**(struct ccw_device * cdev)
switch off the channel measurement for a specific device

Parameters

struct ccw_device * cdev The ccw device to be disabled

Return

0 for success or a negative error value.

Context

non-atomic

u64 **cmf_read**(struct ccw_device * cdev, int index)
read one value from the current channel measurement block

Parameters

struct ccw_device * cdev the channel to be read

int index the index of the value to be read

Return

The value read or `0` if the value cannot be read.

Context

any

int **cmf_readall**(struct ccw_device * cdev, struct cmbdata * data)
read the current channel measurement block

Parameters

struct ccw_device * cdev the channel to be read

struct cmbdata * data a pointer to a data block that will be filled

Return

`0` on success, a negative error value otherwise.

Context

any

45.4 The ccwgroup bus

The ccwgroup bus only contains artificial devices, created by the user. Many networking devices (e.g. qeth) are in fact composed of several ccw devices (like read, write and data channel for qeth). The ccwgroup bus provides a mechanism to create a meta-device which contains those ccw devices as slave devices and can be associated with the netdevice.

45.4.1 ccw group devices

struct **ccwgroup_device**
ccw group device

Definition

```
struct ccwgroup_device {
  enum {
    CCWGROUP_OFFLINE,
    CCWGROUP_ONLINE,
  } state;
  unsigned int count;
  struct device   dev;
  struct work_struct ungroup_work;
  struct ccw_device *cdev[0];
};
```

Members

state online/offline state

count number of attached slave devices

dev embedded device structure

ungroup_work work to be done when a ccwgroup notifier has action type BUS_NOTIFY_UNBIND_DRIVER

cdev variable number of slave devices, allocated as needed

struct **ccwgroup_driver**
driver for ccw group devices

Definition

```
struct ccwgroup_driver {
  int (*setup) (struct ccwgroup_device *);
  void (*remove) (struct ccwgroup_device *);
  int (*set_online) (struct ccwgroup_device *);
  int (*set_offline) (struct ccwgroup_device *);
  void (*shutdown)(struct ccwgroup_device *);
  struct device_driver driver;
  struct ccw_driver *ccw_driver;
};
```

Members

setup function called during device creation to setup the device

remove function called on remove

set_online function called when device is set online

set_offline function called when device is set offline

shutdown function called when device is shut down

driver embedded driver structure

ccw_driver supported ccw_driver (optional)

int **ccwgroup_set_online**(struct ccwgroup_device * gdev)
enable a ccwgroup device

Parameters

struct ccwgroup_device * gdev target ccwgroup device

Description

This function attempts to put the ccwgroup device into the online state.

Return

0 on success and a negative error value on failure.

int **ccwgroup_set_offline**(struct ccwgroup_device * gdev)
disable a ccwgroup device

Parameters

struct ccwgroup_device * gdev target ccwgroup device

Description

This function attempts to put the ccwgroup device into the offline state.

Return

0 on success and a negative error value on failure.

int **ccwgroup_create_dev**(struct device * parent, struct ccwgroup_driver * gdrv, int num_devices, const char * buf)
create and register a ccw group device

Parameters

struct device * parent parent device for the new device

struct ccwgroup_driver * gdrv driver for the new group device

int num_devices number of slave devices

const char * buf buffer containing comma separated bus ids of slave devices

Description

Create and register a new ccw group device as a child of **parent**. Slave devices are obtained from the list of bus ids given in **buf**.

Return

`0` on success and an error code on failure.

Context

non-atomic

int **ccwgroup_driver_register**(struct ccwgroup_driver * cdriver)
register a ccw group driver

Parameters

struct ccwgroup_driver * cdriver driver to be registered

Description

This function is mainly a wrapper around `driver_register()`.

void **ccwgroup_driver_unregister**(struct ccwgroup_driver * cdriver)
deregister a ccw group driver

Parameters

struct ccwgroup_driver * cdriver driver to be deregistered

Description

This function is mainly a wrapper around `driver_unregister()`.

struct ccwgroup_device * **get_ccwgroupdev_by_busid**(struct ccwgroup_driver * gdrv, char * bus_id)
obtain device from a bus id

Parameters

struct ccwgroup_driver * gdrv driver the device is owned by

char * bus_id bus id of the device to be searched

Description

This function searches all devices owned by **gdrv** for a device with a bus id matching **bus_id**.

Return

If a match is found, its reference count of the found device is increased and it is returned; else `NULL` is returned.

int **ccwgroup_probe_ccwdev**(struct ccw_device * cdev)
probe function for slave devices

Parameters

struct ccw_device * cdev ccw device to be probed

Description

This is a dummy probe function for ccw devices that are slave devices in a ccw group device.

Return

always `0`

void **ccwgroup_remove_ccwdev**(struct ccw_device * cdev)
remove function for slave devices

Parameters

struct ccw_device * cdev ccw device to be removed

Description

This is a remove function for ccw devices that are slave devices in a ccw group device. It sets the ccw device offline and also deregisters the embedding ccw group device.

45.5 Generic interfaces

The following section contains interfaces in use not only by drivers dealing with ccw devices, but drivers for various other s390 hardware as well.

45.5.1 Adapter interrupts

The common I/O layer provides helper functions for dealing with adapter interrupts and interrupt vectors.

int **register_adapter_interrupt**(struct airq_struct * airq)
register adapter interrupt handler

Parameters

struct airq_struct * airq pointer to adapter interrupt descriptor

Description

Returns 0 on success, or -EINVAL.

void **unregister_adapter_interrupt**(struct airq_struct * airq)
unregister adapter interrupt handler

Parameters

struct airq_struct * airq pointer to adapter interrupt descriptor

struct airq_iv * **airq_iv_create**(unsigned long bits, unsigned long flags)
create an interrupt vector

Parameters

unsigned long bits number of bits in the interrupt vector

unsigned long flags allocation flags

Description

Returns a pointer to an interrupt vector structure

void **airq_iv_release**(struct airq_iv * iv)
release an interrupt vector

Parameters

struct airq_iv * iv pointer to interrupt vector structure

unsigned long **airq_iv_alloc**(struct airq_iv * iv, unsigned long num)
allocate irq bits from an interrupt vector

Parameters

struct airq_iv * iv pointer to an interrupt vector structure

unsigned long num number of consecutive irq bits to allocate

Description

Returns the bit number of the first irq in the allocated block of irqs, or -1UL if no bit is available or the AIRQ_IV_ALLOC flag has not been specified

void **airq_iv_free**(struct airq_iv * iv, unsigned long bit, unsigned long num)
free irq bits of an interrupt vector

Parameters

struct airq_iv * iv pointer to interrupt vector structure

unsigned long bit number of the first irq bit to free

unsigned long num number of consecutive irq bits to free

unsigned long **airq_iv_scan**(struct airq_iv * iv, unsigned long start, unsigned long end)
scan interrupt vector for non-zero bits

Parameters

struct airq_iv * iv pointer to interrupt vector structure

unsigned long start bit number to start the search

unsigned long end bit number to end the search

Description

Returns the bit number of the next non-zero interrupt bit, or -1UL if the scan completed without finding any more any non-zero bits.

CHAPTER

FORTYSIX

VME DEVICE DRIVERS

46.1 Driver registration

As with other subsystems within the Linux kernel, VME device drivers register with the VME subsystem, typically called from the devices init routine. This is achieved via a call to `vme_register_driver()`.

A pointer to a structure of type `struct vme_driver` must be provided to the registration function. Along with the maximum number of devices your driver is able to support.

At the minimum, the '.name', '.match' and '.probe' elements of `struct vme_driver` should be correctly set. The '.name' element is a pointer to a string holding the device driver's name.

The '.match' function allows control over which VME devices should be registered with the driver. The match function should return 1 if a device should be probed and 0 otherwise. This example match function (from vme_user.c) limits the number of devices probed to one:

```
#define USER_BUS_MAX    1
...
static int vme_user_match(struct vme_dev *vdev)
{
        if (vdev->id.num >= USER_BUS_MAX)
                return 0;
        return 1;
}
```

The '.probe' element should contain a pointer to the probe routine. The probe routine is passed a `struct vme_dev` pointer as an argument.

Here, the 'num' field refers to the sequential device ID for this specific driver. The bridge number (or bus number) can be accessed using dev->bridge->num.

A function is also provided to unregister the driver from the VME core called `vme_unregister_driver()` and should usually be called from the device driver's exit routine.

46.2 Resource management

Once a driver has registered with the VME core the provided match routine will be called the number of times specified during the registration. If a match succeeds, a non-zero value should be returned. A zero return value indicates failure. For all successful matches, the probe routine of the corresponding driver is called. The probe routine is passed a pointer to the devices device structure. This pointer should be saved, it will be required for requesting VME resources.

The driver can request ownership of one or more master windows (`vme_master_request()`), slave windows (`vme_slave_request()`) and/or dma channels (`vme_dma_request()`). Rather than allowing the device driver to request a specific window or DMA channel (which may be used by a different driver) the API allows a resource to be assigned based on the required attributes of the driver in question. For slave windows these attributes are split into the VME address spaces that need to be accessed in 'aspace' and VME bus cycle types required in 'cycle'. Master windows add a further set of attributes in 'width' specifying the required data transfer widths. These attributes are defined as bitmasks and as such any combination of the attributes can be requested for a single window, the core will assign a window that meets the requirements, returning a pointer of type vme_resource that should be used to identify the allocated resource when it is used. For DMA controllers, the request function requires the potential direction of any transfers to be provided in the route attributes. This is typically VME-to-MEM and/or MEM-to-VME, though some hardware can support VME-to-VME and MEM-to-MEM transfers as well as test pattern generation. If an unallocated window fitting the requirements can not be found a NULL pointer will be returned.

Functions are also provided to free window allocations once they are no longer required. These functions (`vme_master_free()`, `vme_slave_free()` and `vme_dma_free()`) should be passed the pointer to the resource provided during resource allocation.

46.3 Master windows

Master windows provide access from the local processor[s] out onto the VME bus. The number of windows available and the available access modes is dependent on the underlying chipset. A window must be configured before it can be used.

46.3.1 Master window configuration

Once a master window has been assigned `vme_master_set()` can be used to configure it and `vme_master_get()` to retrieve the current settings. The address spaces, transfer widths and cycle types are the same as described under resource management, however some of the options are mutually exclusive. For example, only one address space may be specified.

46.3.2 Master window access

The function `vme_master_read()` can be used to read from and `vme_master_write()` used to write to configured master windows.

In addition to simple reads and writes, `vme_master_rmw()` is provided to do a read-modify-write transaction. Parts of a VME window can also be mapped into user space memory using `vme_master_mmap()`.

46.4 Slave windows

Slave windows provide devices on the VME bus access into mapped portions of the local memory. The number of windows available and the access modes that can be used is dependent on the underlying chipset. A window must be configured before it can be used.

46.4.1 Slave window configuration

Once a slave window has been assigned `vme_slave_set()` can be used to configure it and `vme_slave_get()` to retrieve the current settings.

The address spaces, transfer widths and cycle types are the same as described under resource management, however some of the options are mutually exclusive. For example, only one address space may be specified.

46.4.2 Slave window buffer allocation

Functions are provided to allow the user to allocate (`vme_alloc_consistent()`) and free (`vme_free_consistent()`) contiguous buffers which will be accessible by the VME bridge. These functions do not have to be used, other methods can be used to allocate a buffer, though care must be taken to ensure that they are contiguous and accessible by the VME bridge.

46.4.3 Slave window access

Slave windows map local memory onto the VME bus, the standard methods for accessing memory should be used.

46.5 DMA channels

The VME DMA transfer provides the ability to run link-list DMA transfers. The API introduces the concept of DMA lists. Each DMA list is a link-list which can be passed to a DMA controller. Multiple lists can be created, extended, executed, reused and destroyed.

46.5.1 List Management

The function `vme_new_dma_list()` is provided to create and `vme_dma_list_free()` to destroy DMA lists. Execution of a list will not automatically destroy the list, thus enabling a list to be reused for repetitive tasks.

46.5.2 List Population

An item can be added to a list using `vme_dma_list_add()` (the source and destination attributes need to be created before calling this function, this is covered under "Transfer Attributes").

Note: The detailed attributes of the transfers source and destination are not checked until an entry is added to a DMA list, the request for a DMA channel purely checks the directions in which the controller is expected to transfer data. As a result it is possible for this call to return an error, for example if the source or destination is in an unsupported VME address space.

46.5.3 Transfer Attributes

The attributes for the source and destination are handled separately from adding an item to a list. This is due to the diverse attributes required for each type of source and destination. There are functions to create attributes for PCI, VME and pattern sources and destinations (where appropriate):

- PCI source or destination: `vme_dma_pci_attribute()`
- VME source or destination: `vme_dma_vme_attribute()`
- Pattern source: `vme_dma_pattern_attribute()`

The function `vme_dma_free_attribute()` should be used to free an attribute.

46.5.4 List Execution

The function `vme_dma_list_exec()` queues a list for execution and will return once the list has been executed.

46.6 Interrupts

The VME API provides functions to attach and detach callbacks to specific VME level and status ID combinations and for the generation of VME interrupts with specific VME level and status IDs.

46.6.1 Attaching Interrupt Handlers

The function `vme_irq_request()` can be used to attach and `vme_irq_free()` to free a specific VME level and status ID combination. Any given combination can only be assigned a single callback function. A void pointer parameter is provided, the value of which is passed to the callback function, the use of this pointer is user undefined. The callback parameters are as follows. Care must be taken in writing a callback function, callback functions run in interrupt context:

```
void callback(int level, int statid, void *priv);
```

46.6.2 Interrupt Generation

The function `vme_irq_generate()` can be used to generate a VME interrupt at a given VME level and VME status ID.

46.7 Location monitors

The VME API provides the following functionality to configure the location monitor.

46.7.1 Location Monitor Management

The function `vme_lm_request()` is provided to request the use of a block of location monitors and `vme_lm_free()` to free them after they are no longer required. Each block may provide a number of location monitors, monitoring adjacent locations. The function `vme_lm_count()` can be used to determine how many locations are provided.

46.7.2 Location Monitor Configuration

Once a bank of location monitors has been allocated, the function `vme_lm_set()` is provided to configure the location and mode of the location monitor. The function `vme_lm_get()` can be used to retrieve existing settings.

46.7.3 Location Monitor Use

The function `vme_lm_attach()` enables a callback to be attached and `vme_lm_detach()` allows on to be detached from each location monitor location. Each location monitor can monitor a number of adjacent locations. The callback function is declared as follows.

```
void callback(void *data);
```

46.8 Slot Detection

The function `vme_slot_num()` returns the slot ID of the provided bridge.

46.9 Bus Detection

The function `vme_bus_num()` returns the bus ID of the provided bridge.

46.10 VME API

struct **vme_dev**
Structure representing a VME device

Definition

```
struct vme_dev {
  int num;
  struct vme_bridge *bridge;
  struct device dev;
  struct list_head drv_list;
  struct list_head bridge_list;
};
```

Members

num The device number

bridge Pointer to the bridge device this device is on

dev Internal device structure

drv_list List of devices (per driver)

bridge_list List of devices (per bridge)

struct **vme_driver**
Structure representing a VME driver

Definition

```
struct vme_driver {
  const char *name;
  int (*match)(struct vme_dev *);
  int (*probe)(struct vme_dev *);
  int (*remove)(struct vme_dev *);
  struct device_driver driver;
  struct list_head devices;
};
```

Members

name Driver name, should be unique among VME drivers and usually the same as the module name.

match Callback used to determine whether probe should be run.

probe Callback for device binding, called when new device is detected.

remove Callback, called on device removal.

driver Underlying generic device driver structure.

devices List of VME devices (struct vme_dev) associated with this driver.

void * **vme_alloc_consistent**(struct vme_resource * resource, size_t size, dma_addr_t * dma)
Allocate contiguous memory.

Parameters

struct vme_resource * resource Pointer to VME resource.

size_t size Size of allocation required.

dma_addr_t * dma Pointer to variable to store physical address of allocation.

Description

Allocate a contiguous block of memory for use by the driver. This is used to create the buffers for the slave windows.

Return

Virtual address of allocation on success, NULL on failure.

void **vme_free_consistent**(struct vme_resource * resource, size_t size, void * vaddr, dma_addr_t dma)
Free previously allocated memory.

Parameters

struct vme_resource * resource Pointer to VME resource.

size_t size Size of allocation to free.

void * vaddr Virtual address of allocation.

dma_addr_t dma Physical address of allocation.

Description

Free previously allocated block of contiguous memory.

size_t **vme_get_size**(struct vme_resource * resource)
Helper function returning size of a VME window

Parameters

struct vme_resource * resource Pointer to VME slave or master resource.

Description

Determine the size of the VME window provided. This is a helper function, wrappering the call to vme_master_get or vme_slave_get depending on the type of window resource handed to it.

Return

Size of the window on success, zero on failure.

struct vme_resource * **vme_slave_request**(struct vme_dev * vdev, u32 address, u32 cycle)
Request a VME slave window resource.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

u32 address Required VME address space.

u32 cycle Required VME data transfer cycle type.

Description

Request use of a VME window resource capable of being set for the requested address space and data transfer cycle.

Return

Pointer to VME resource on success, NULL on failure.

int **vme_slave_set**(struct vme_resource * resource, int enabled, unsigned long long vme_base, unsigned long long size, dma_addr_t buf_base, u32 aspace, u32 cycle)
Set VME slave window configuration.

Parameters

struct vme_resource * resource Pointer to VME slave resource.

int enabled State to which the window should be configured.

unsigned long long vme_base Base address for the window.

unsigned long long size Size of the VME window.

dma_addr_t buf_base Based address of buffer used to provide VME slave window storage.

u32 aspace VME address space for the VME window.

u32 cycle VME data transfer cycle type for the VME window.

Description

Set configuration for provided VME slave window.

Return

Zero on success, -EINVAL if operation is not supported on this device, if an invalid resource has been provided or invalid attributes are provided. Hardware specific errors may also be returned.

int **vme_slave_get**(struct vme_resource * resource, int * enabled, unsigned long long * vme_base, unsigned long long * size, dma_addr_t * buf_base, u32 * aspace, u32 * cycle)
Retrieve VME slave window configuration.

Parameters

struct vme_resource * resource Pointer to VME slave resource.

int * enabled Pointer to variable for storing state.

unsigned long long * vme_base Pointer to variable for storing window base address.

unsigned long long * size Pointer to variable for storing window size.

dma_addr_t * buf_base Pointer to variable for storing slave buffer base address.

u32 * aspace Pointer to variable for storing VME address space.

u32 * cycle Pointer to variable for storing VME data transfer cycle type.

Description

Return configuration for provided VME slave window.

Return

Zero on success, -EINVAL if operation is not supported on this device or if an invalid resource has been provided.

void **vme_slave_free**(struct vme_resource * resource)
Free VME slave window

Parameters

struct vme_resource * resource Pointer to VME slave resource.

Description

Free the provided slave resource so that it may be reallocated.

struct vme_resource * **vme_master_request**(struct vme_dev * vdev, u32 address, u32 cycle, u32 dwidth)
Request a VME master window resource.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

u32 address Required VME address space.

u32 cycle Required VME data transfer cycle type.

u32 dwidth Required VME data transfer width.

Description

Request use of a VME window resource capable of being set for the requested address space, data transfer cycle and width.

Return

Pointer to VME resource on success, NULL on failure.

int **vme_master_set**(struct vme_resource * resource, int enabled, unsigned long long vme_base, unsigned long long size, u32 aspace, u32 cycle, u32 dwidth)
Set VME master window configuration.

Parameters

struct vme_resource * resource Pointer to VME master resource.

int enabled State to which the window should be configured.

unsigned long long vme_base Base address for the window.

unsigned long long size Size of the VME window.

u32 aspace VME address space for the VME window.

u32 cycle VME data transfer cycle type for the VME window.

u32 dwidth VME data transfer width for the VME window.

Description

Set configuration for provided VME master window.

Return

Zero on success, -EINVAL if operation is not supported on this device, if an invalid resource has been provided or invalid attributes are provided. Hardware specific errors may also be returned.

int **vme_master_get**(struct vme_resource * resource, int * enabled, unsigned long long * vme_base, unsigned long long * size, u32 * aspace, u32 * cycle, u32 * dwidth)

Retrieve VME master window configuration.

Parameters

struct vme_resource * resource Pointer to VME master resource.

int * enabled Pointer to variable for storing state.

unsigned long long * vme_base Pointer to variable for storing window base address.

unsigned long long * size Pointer to variable for storing window size.

u32 * aspace Pointer to variable for storing VME address space.

u32 * cycle Pointer to variable for storing VME data transfer cycle type.

u32 * dwidth Pointer to variable for storing VME data transfer width.

Description

Return configuration for provided VME master window.

Return

Zero on success, -EINVAL if operation is not supported on this device or if an invalid resource has been provided.

ssize_t **vme_master_read**(struct vme_resource * resource, void * buf, size_t count, loff_t offset)

Read data from VME space into a buffer.

Parameters

struct vme_resource * resource Pointer to VME master resource.

void * buf Pointer to buffer where data should be transferred.

size_t count Number of bytes to transfer.

loff_t offset Offset into VME master window at which to start transfer.

Description

Perform read of count bytes of data from location on VME bus which maps into the VME master window at offset to buf.

Return

Number of bytes read, -EINVAL if resource is not a VME master resource or read operation is not supported. -EFAULT returned if invalid offset is provided. Hardware specific errors may also be returned.

ssize_t **vme_master_write**(struct vme_resource * resource, void * buf, size_t count, loff_t offset)

Write data out to VME space from a buffer.

Parameters

struct vme_resource * resource Pointer to VME master resource.

void * buf Pointer to buffer holding data to transfer.

size_t count Number of bytes to transfer.

loff_t offset Offset into VME master window at which to start transfer.

Description

Perform write of count bytes of data from buf to location on VME bus which maps into the VME master window at offset.

Return

Number of bytes written, -EINVAL if resource is not a VME master resource or write operation is not supported. -EFAULT returned if invalid offset is provided. Hardware specific errors may also be returned.

unsigned int **vme_master_rmw**(struct vme_resource * resource, unsigned int mask, unsigned int compare, unsigned int swap, loff_t offset)

Perform read-modify-write cycle.

Parameters

struct vme_resource * resource Pointer to VME master resource.

unsigned int mask Bits to be compared and swapped in operation.

unsigned int compare Bits to be compared with data read from offset.

unsigned int swap Bits to be swapped in data read from offset.

loff_t offset Offset into VME master window at which to perform operation.

Description

Perform read-modify-write cycle on provided location: - Location on VME bus is read. - Bits selected by mask are compared with compare. - Where a selected bit matches that in compare and are selected in swap, the bit is swapped. - Result written back to location on VME bus.

Return

Bytes written on success, -EINVAL if resource is not a VME master resource or RMW operation is not supported. Hardware specific errors may also be returned.

int **vme_master_mmap**(struct vme_resource * resource, struct vm_area_struct * vma)
Mmap region of VME master window.

Parameters

struct vme_resource * resource Pointer to VME master resource.

struct vm_area_struct * vma Pointer to definition of user mapping.

Description

Memory map a region of the VME master window into user space.

Return

Zero on success, -EINVAL if resource is not a VME master resource or -EFAULT if map exceeds window size. Other generic mmap errors may also be returned.

void **vme_master_free**(struct vme_resource * resource)
Free VME master window

Parameters

struct vme_resource * resource Pointer to VME master resource.

Description

Free the provided master resource so that it may be reallocated.

struct vme_resource * **vme_dma_request**(struct vme_dev * vdev, u32 route)
Request a DMA controller.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

u32 route Required src/destination combination.

Description

Request a VME DMA controller with capability to perform transfers bewteen requested source/destination combination.

Return

Pointer to VME DMA resource on success, NULL on failure.

struct vme_dma_list * **vme_new_dma_list**(struct vme_resource * resource)
Create new VME DMA list.

Parameters

struct vme_resource * resource Pointer to VME DMA resource.

Description

Create a new VME DMA list. It is the responsibility of the user to free the list once it is no longer required with `vme_dma_list_free()`.

Return

Pointer to new VME DMA list, NULL on allocation failure or invalid VME DMA resource.

struct vme_dma_attr * **vme_dma_pattern_attribute**(u32 pattern, u32 type)
Create "Pattern" type VME DMA list attribute.

Parameters

u32 pattern Value to use used as pattern

u32 type Type of pattern to be written.

Description

Create VME DMA list attribute for pattern generation. It is the responsibility of the user to free used attributes using `vme_dma_free_attribute()`.

Return

Pointer to VME DMA attribute, NULL on failure.

struct vme_dma_attr * **vme_dma_pci_attribute**(dma_addr_t address)
Create "PCI" type VME DMA list attribute.

Parameters

dma_addr_t address PCI base address for DMA transfer.

Description

Create VME DMA list attribute pointing to a location on PCI for DMA transfers. It is the responsibility of the user to free used attributes using `vme_dma_free_attribute()`.

Return

Pointer to VME DMA attribute, NULL on failure.

struct vme_dma_attr * **vme_dma_vme_attribute**(unsigned long long address, u32 aspace, u32 cycle, u32 dwidth)
Create "VME" type VME DMA list attribute.

Parameters

unsigned long long address VME base address for DMA transfer.

u32 aspace VME address space to use for DMA transfer.

u32 cycle VME bus cycle to use for DMA transfer.

u32 dwidth VME data width to use for DMA transfer.

Description

Create VME DMA list attribute pointing to a location on the VME bus for DMA transfers. It is the responsibility of the user to free used attributes using `vme_dma_free_attribute()`.

Return

Pointer to VME DMA attribute, NULL on failure.

void **vme_dma_free_attribute**(struct vme_dma_attr * attributes)
Free DMA list attribute.

Parameters

struct vme_dma_attr * attributes Pointer to DMA list attribute.

Description

Free VME DMA list attribute. VME DMA list attributes can be safely freed once `vme_dma_list_add()` has returned.

int **vme_dma_list_add**(struct vme_dma_list * list, struct vme_dma_attr * src,
struct vme_dma_attr * dest, size_t count)
Add enty to a VME DMA list.

Parameters

struct vme_dma_list * list Pointer to VME list.

struct vme_dma_attr * src Pointer to DMA list attribute to use as source.

struct vme_dma_attr * dest Pointer to DMA list attribute to use as destination.

size_t count Number of bytes to transfer.

Description

Add an entry to the provided VME DMA list. Entry requires pointers to source and destination DMA attributes and a count.

Please note, the attributes supported as source and destinations for transfers are hardware dependent.

Return

Zero on success, -EINVAL if operation is not supported on this device or if the link list has already been submitted for execution. Hardware specific errors also possible.

int **vme_dma_list_exec**(struct vme_dma_list * list)
Queue a VME DMA list for execution.

Parameters

struct vme_dma_list * list Pointer to VME list.

Description

Queue the provided VME DMA list for execution. The call will return once the list has been executed.

Return

Zero on success, -EINVAL if operation is not supported on this device. Hardware specific errors also possible.

int **vme_dma_list_free**(struct vme_dma_list * list)
Free a VME DMA list.

Parameters

struct vme_dma_list * list Pointer to VME list.

Description

Free the provided DMA list and all its entries.

Return

Zero on success, -EINVAL on invalid VME resource, -EBUSY if resource is still in use. Hardware specific errors also possible.

int **vme_dma_free**(struct vme_resource * resource)
Free a VME DMA resource.

Parameters

struct vme_resource * resource Pointer to VME DMA resource.

Description

Free the provided DMA resource so that it may be reallocated.

Return

Zero on success, -EINVAL on invalid VME resource, -EBUSY if resource is still active.

int **vme_irq_request**(struct vme_dev * vdev, int level, int statid, void (*callback)(int, int, void *), void * priv_data)
Request a specific VME interrupt.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

int level Interrupt priority being requested.

int statid Interrupt vector being requested.

void (*)(int, int, void *) callback Pointer to callback function called when VME interrupt/vector received.

void * priv_data Generic pointer that will be passed to the callback function.

Description

Request callback to be attached as a handler for VME interrupts with provided level and statid.

Return

Zero on success, -EINVAL on invalid vme device, level or if the function is not supported, -EBUSY if the level/statid combination is already in use. Hardware specific errors also possible.

void **vme_irq_free**(struct vme_dev * vdev, int level, int statid)
Free a VME interrupt.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

int level Interrupt priority of interrupt being freed.

int statid Interrupt vector of interrupt being freed.

Description

Remove previously attached callback from VME interrupt priority/vector.

int **vme_irq_generate**(struct vme_dev * vdev, int level, int statid)
Generate VME interrupt.

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

int level Interrupt priority at which to assert the interrupt.

int statid Interrupt vector to associate with the interrupt.

Description

Generate a VME interrupt of the provided level and with the provided statid.

Return

Zero on success, -EINVAL on invalid vme device, level or if the function is not supported. Hardware specific errors also possible.

struct vme_resource * **vme_lm_request**(struct vme_dev * vdev)
Request a VME location monitor

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

Description

Allocate a location monitor resource to the driver. A location monitor allows the driver to monitor accesses to a contiguous number of addresses on the VME bus.

Return

Pointer to a VME resource on success or NULL on failure.

int **vme_lm_count**(struct vme_resource * resource)
Determine number of VME Addresses monitored

Parameters

struct vme_resource * resource Pointer to VME location monitor resource.

Description

The number of contiguous addresses monitored is hardware dependent. Return the number of contiguous addresses monitored by the location monitor.

Return

Count of addresses monitored or -EINVAL when provided with an invalid location monitor resource.

int **vme_lm_set**(struct vme_resource * resource, unsigned long long lm_base, u32 aspace, u32 cycle)
Configure location monitor

Parameters

struct vme_resource * resource Pointer to VME location monitor resource.

unsigned long long lm_base Base address to monitor.

u32 aspace VME address space to monitor.

u32 cycle VME bus cycle type to monitor.

Description

Set the base address, address space and cycle type of accesses to be monitored by the location monitor.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource or function is not supported. Hardware specific errors may also be returned.

int **vme_lm_get**(struct vme_resource * resource, unsigned long long * lm_base, u32 * aspace, u32 * cycle)
Retrieve location monitor settings

Parameters

struct vme_resource * resource Pointer to VME location monitor resource.

unsigned long long * lm_base Pointer used to output the base address monitored.

u32 * aspace Pointer used to output the address space monitored.

u32 * cycle Pointer used to output the VME bus cycle type monitored.

Description

Retrieve the base address, address space and cycle type of accesses to be monitored by the location monitor.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource or function is not supported. Hardware specific errors may also be returned.

int **vme_lm_attach**(struct vme_resource * resource, int monitor, void (*callback)(void *), void * data)
Provide callback for location monitor address

Parameters

struct vme_resource * resource Pointer to VME location monitor resource.

int monitor Offset to which callback should be attached.

void (*)(void *) callback Pointer to callback function called when triggered.

void * data Generic pointer that will be passed to the callback function.

Description

Attach a callback to the specificed offset into the location monitors monitored addresses. A generic pointer is provided to allow data to be passed to the callback when called.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource or function is not supported. Hardware specific errors may also be returned.

int **vme_lm_detach**(struct vme_resource * resource, int monitor)
Remove callback for location monitor address

Parameters

struct vme_resource * resource Pointer to VME location monitor resource.

int monitor Offset to which callback should be removed.

Description

Remove the callback associated with the specificed offset into the location monitors monitored addresses.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource or function is not supported. Hardware specific errors may also be returned.

void **vme_lm_free**(struct vme_resource * resource)
Free allocated VME location monitor

Parameters

struct vme_resource * resource Pointer to VME location monitor resource.

Description

Free allocation of a VME location monitor.

WARNING: This function currently expects that any callbacks that have been attached to the location monitor have been removed.

Return

Zero on success, -EINVAL when provided with an invalid location monitor resource.

int **vme_slot_num**(struct vme_dev * vdev)
Retrieve slot ID

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

Description

Retrieve the slot ID associated with the provided VME device.

Return

The slot ID on success, -EINVAL if VME bridge cannot be determined or the function is not supported. Hardware specific errors may also be returned.

int **vme_bus_num**(struct vme_dev * vdev)
Retrieve bus number

Parameters

struct vme_dev * vdev Pointer to VME device struct vme_dev assigned to driver instance.

Description

Retrieve the bus enumeration associated with the provided VME device.

Return

The bus number on success, -EINVAL if VME bridge cannot be determined.

int **vme_register_driver**(struct vme_driver * drv, unsigned int ndevs)
Register a VME driver

Parameters

struct vme_driver * drv Pointer to VME driver structure to register.

unsigned int ndevs Maximum number of devices to allow to be enumerated.

Description

Register a VME device driver with the VME subsystem.

Return

Zero on success, error value on registration failure.

void **vme_unregister_driver**(struct vme_driver * drv)
Unregister a VME driver

Parameters

struct vme_driver * drv Pointer to VME driver structure to unregister.

Description

Unregister a VME device driver from the VME subsystem.

CHAPTER

FORTYSEVEN

LINUX 802.11 DRIVER DEVELOPER'S GUIDE

47.1 Introduction

Explaining wireless 802.11 networking in the Linux kernel

These books attempt to give a description of the various subsystems that play a role in 802.11 wireless networking in Linux. Since these books are for kernel developers they attempts to document the structures and functions used in the kernel as well as giving a higher-level overview.

The reader is expected to be familiar with the 802.11 standard as published by the IEEE in 802.11-2007 (or possibly later versions). References to this standard will be given as "802.11-2007 8.1.5".

47.2 cfg80211 subsystem

cfg80211 is the configuration API for 802.11 devices in Linux. It bridges userspace and drivers, and offers some utility functionality associated with 802.11. cfg80211 must, directly or indirectly via mac80211, be used by all modern wireless drivers in Linux, so that they offer a consistent API through nl80211. For backward compatibility, cfg80211 also offers wireless extensions to userspace, but hides them from drivers completely.

Additionally, cfg80211 contains code to help enforce regulatory spectrum use restrictions.

47.2.1 Device registration

In order for a driver to use cfg80211, it must register the hardware device with cfg80211. This happens through a number of hardware capability structs described below.

The fundamental structure for each device is the 'wiphy', of which each instance describes a physical wireless device connected to the system. Each such wiphy can have zero, one, or many virtual interfaces associated with it, which need to be identified as such by pointing the network interface's **ieee80211_ptr** pointer to a `struct wireless_dev` which further describes the wireless part of the interface, normally this struct is embedded in the network interface's private data area.

Drivers can optionally allow creating or destroying virtual interfaces on the fly, but without at least one or the ability to create some the wireless device isn't useful.

Each wiphy structure contains device capability information, and also has a pointer to the various operations the driver offers. The definitions and structures here describe these capabilities in detail.

enum **ieee80211_channel_flags**
channel flags

Constants

IEEE80211_CHAN_DISABLED This channel is disabled.

IEEE80211_CHAN_NO_IR do not initiate radiation, this includes sending probe requests or beaconing.

IEEE80211_CHAN_RADAR Radar detection is required on this channel.

IEEE80211_CHAN_NO_HT40PLUS extension channel above this channel is not permitted.

IEEE80211_CHAN_NO_HT40MINUS extension channel below this channel is not permitted.

IEEE80211_CHAN_NO_OFDM OFDM is not allowed on this channel.

IEEE80211_CHAN_NO_80MHZ If the driver supports 80 MHz on the band, this flag indicates that an 80 MHz channel cannot use this channel as the control or any of the secondary channels. This may be due to the driver or due to regulatory bandwidth restrictions.

IEEE80211_CHAN_NO_160MHZ If the driver supports 160 MHz on the band, this flag indicates that an 160 MHz channel cannot use this channel as the control or any of the secondary channels. This may be due to the driver or due to regulatory bandwidth restrictions.

IEEE80211_CHAN_INDOOR_ONLY see `NL80211_FREQUENCY_ATTR_INDOOR_ONLY`

IEEE80211_CHAN_IR_CONCURRENT see `NL80211_FREQUENCY_ATTR_IR_CONCURRENT`

IEEE80211_CHAN_NO_20MHZ 20 MHz bandwidth is not permitted on this channel.

IEEE80211_CHAN_NO_10MHZ 10 MHz bandwidth is not permitted on this channel.

IEEE80211_CHAN_NO_HE HE operation is not permitted on this channel.

Description

Channel flags set by the regulatory control code.

struct **ieee80211_channel**
channel definition

Definition

```
struct ieee80211_channel {
  enum nl80211_band band;
  u32 center_freq;
  u16 freq_offset;
```

(continues on next page)

(continued from previous page)

```
  u16 hw_value;
  u32 flags;
  int max_antenna_gain;
  int max_power;
  int max_reg_power;
  bool beacon_found;
  u32 orig_flags;
  int orig_mag, orig_mpwr;
  enum nl80211_dfs_state dfs_state;
  unsigned long dfs_state_entered;
  unsigned int dfs_cac_ms;
};
```

Members

band band this channel belongs to.

center_freq center frequency in MHz

freq_offset offset from **center_freq**, in KHz

hw_value hardware-specific value for the channel

flags channel flags from `enum ieee80211_channel_flags`.

max_antenna_gain maximum antenna gain in dBi

max_power maximum transmission power (in dBm)

max_reg_power maximum regulatory transmission power (in dBm)

beacon_found helper to regulatory code to indicate when a beacon has been found on this channel. Use regulatory_hint_found_beacon() to enable this, this is useful only on 5 GHz band.

orig_flags channel flags at registration time, used by regulatory code to support devices with additional restrictions

orig_mag internal use

orig_mpwr internal use

dfs_state current state of this channel. Only relevant if radar is required on this channel.

dfs_state_entered timestamp (jiffies) when the dfs state was entered.

dfs_cac_ms DFS CAC time in milliseconds, this is valid for DFS channels.

Description

This structure describes a single channel for use with cfg80211.

enum **ieee80211_rate_flags**
rate flags

Constants

IEEE80211_RATE_SHORT_PREAMBLE Hardware can send with short preamble on this bitrate; only relevant in 2.4GHz band and with CCK rates.

IEEE80211_RATE_MANDATORY_A This bitrate is a mandatory rate when used with 802.11a (on the 5 GHz band); filled by the core code when registering the wiphy.

IEEE80211_RATE_MANDATORY_B This bitrate is a mandatory rate when used with 802.11b (on the 2.4 GHz band); filled by the core code when registering the wiphy.

IEEE80211_RATE_MANDATORY_G This bitrate is a mandatory rate when used with 802.11g (on the 2.4 GHz band); filled by the core code when registering the wiphy.

IEEE80211_RATE_ERP_G This is an ERP rate in 802.11g mode.

IEEE80211_RATE_SUPPORTS_5MHZ Rate can be used in 5 MHz mode

IEEE80211_RATE_SUPPORTS_10MHZ Rate can be used in 10 MHz mode

Description

Hardware/specification flags for rates. These are structured in a way that allows using the same bitrate structure for different bands/PHY modes.

struct **ieee80211_rate**
bitrate definition

Definition

```
struct ieee80211_rate {
  u32 flags;
  u16 bitrate;
  u16 hw_value, hw_value_short;
};
```

Members

flags rate-specific flags

bitrate bitrate in units of 100 Kbps

hw_value driver/hardware value for this rate

hw_value_short driver/hardware value for this rate when short preamble is used

Description

This structure describes a bitrate that an 802.11 PHY can operate with. The two values **hw_value** and **hw_value_short** are only for driver use when pointers to this structure are passed around.

struct **ieee80211_sta_ht_cap**
STA's HT capabilities

Definition

```
struct ieee80211_sta_ht_cap {
  u16 cap;
  bool ht_supported;
  u8 ampdu_factor;
  u8 ampdu_density;
```

(continues on next page)

(continued from previous page)

```
  struct ieee80211_mcs_info mcs;
};
```

Members

cap HT capabilities map as described in 802.11n spec

ht_supported is HT supported by the STA

ampdu_factor Maximum A-MPDU length factor

ampdu_density Minimum A-MPDU spacing

mcs Supported MCS rates

Description

This structure describes most essential parameters needed to describe 802.11n HT capabilities for an STA.

struct **ieee80211_supported_band**
frequency band definition

Definition

```
struct ieee80211_supported_band {
  struct ieee80211_channel *channels;
  struct ieee80211_rate *bitrates;
  enum nl80211_band band;
  int n_channels;
  int n_bitrates;
  struct ieee80211_sta_ht_cap ht_cap;
  struct ieee80211_sta_vht_cap vht_cap;
  struct ieee80211_edmg edmg_cap;
  u16 n_iftype_data;
  const struct ieee80211_sband_iftype_data *iftype_data;
};
```

Members

channels Array of channels the hardware can operate in in this band.

bitrates Array of bitrates the hardware can operate with in this band. Must be sorted to give a valid "supported rates" IE, i.e. CCK rates first, then OFDM.

band the band this structure represents

n_channels Number of channels in **channels**

n_bitrates Number of bitrates in **bitrates**

ht_cap HT capabilities in this band

vht_cap VHT capabilities in this band

edmg_cap EDMG capabilities in this band

n_iftype_data number of iftype data entries

iftype_data interface type data entries. Note that the bits in **types_mask** inside this structure cannot overlap (i.e. only one occurrence of each type is allowed across all instances of iftype_data).

Description

This structure describes a frequency band a wiphy is able to operate in.

enum **cfg80211_signal_type**
signal type

Constants

CFG80211_SIGNAL_TYPE_NONE no signal strength information available

CFG80211_SIGNAL_TYPE_MBM signal strength in mBm (100*dBm)

CFG80211_SIGNAL_TYPE_UNSPEC signal strength, increasing from 0 through 100

enum **wiphy_params_flags**
set_wiphy_params bitfield values

Constants

WIPHY_PARAM_RETRY_SHORT wiphy->retry_short has changed

WIPHY_PARAM_RETRY_LONG wiphy->retry_long has changed

WIPHY_PARAM_FRAG_THRESHOLD wiphy->frag_threshold has changed

WIPHY_PARAM_RTS_THRESHOLD wiphy->rts_threshold has changed

WIPHY_PARAM_COVERAGE_CLASS coverage class changed

WIPHY_PARAM_DYN_ACK dynack has been enabled

WIPHY_PARAM_TXQ_LIMIT TXQ packet limit has been changed

WIPHY_PARAM_TXQ_MEMORY_LIMIT TXQ memory limit has been changed

WIPHY_PARAM_TXQ_QUANTUM TXQ scheduler quantum

enum **wiphy_flags**
wiphy capability flags

Constants

WIPHY_FLAG_SUPPORTS_EXT_KEK_KCK The device supports bigger kek and kck keys

WIPHY_FLAG_NETNS_OK if not set, do not allow changing the netns of this wiphy at all

WIPHY_FLAG_PS_ON_BY_DEFAULT if set to true, powersave will be enabled by default – this flag will be set depending on the kernel's default on `wiphy_new()`, but can be changed by the driver if it has a good reason to override the default

WIPHY_FLAG_4ADDR_AP supports 4addr mode even on AP (with a single station on a VLAN interface). This flag also serves an extra purpose of supporting 4ADDR AP mode on devices which do not support AP/VLAN iftype.

WIPHY_FLAG_4ADDR_STATION supports 4addr mode even as a station

WIPHY_FLAG_CONTROL_PORT_PROTOCOL This device supports setting the control port protocol ethertype. The device also honours the control_port_no_encrypt flag.

WIPHY_FLAG_IBSS_RSN The device supports IBSS RSN.

WIPHY_FLAG_MESH_AUTH The device supports mesh authentication by routing auth frames to userspace. See **NL80211_MESH_SETUP_USERSPACE_AUTH**.

WIPHY_FLAG_SUPPORTS_FW_ROAM The device supports roaming feature in the firmware.

WIPHY_FLAG_AP_UAPSD The device supports uapsd on AP.

WIPHY_FLAG_SUPPORTS_TDLS The device supports TDLS (802.11z) operation.

WIPHY_FLAG_TDLS_EXTERNAL_SETUP The device does not handle TDLS (802.11z) link setup/discovery operations internally. Setup, discovery and teardown packets should be sent through the **NL80211_CMD_TDLS_MGMT** command. When this flag is not set, **NL80211_CMD_TDLS_OPER** should be used for asking the driver/firmware to perform a TDLS operation.

WIPHY_FLAG_HAVE_AP_SME device integrates AP SME

WIPHY_FLAG_REPORTS_OBSS the device will report beacons from other BSSes when there are virtual interfaces in AP mode by calling cfg80211_report_obss_beacon().

WIPHY_FLAG_AP_PROBE_RESP_OFFLOAD When operating as an AP, the device responds to probe-requests in hardware.

WIPHY_FLAG_OFFCHAN_TX Device supports direct off-channel TX.

WIPHY_FLAG_HAS_REMAIN_ON_CHANNEL Device supports remain-on-channel call.

WIPHY_FLAG_SUPPORTS_5_10_MHZ Device supports 5 MHz and 10 MHz channels.

WIPHY_FLAG_HAS_CHANNEL_SWITCH Device supports channel switch in beaconing mode (AP, IBSS, Mesh, ⋯).

WIPHY_FLAG_HAS_STATIC_WEP The device supports static WEP key installation before connection.

struct **wiphy**
wireless hardware description

Definition

```
struct wiphy {
  u8 perm_addr[ETH_ALEN];
  u8 addr_mask[ETH_ALEN];
  struct mac_address *addresses;
  const struct ieee80211_txrx_stypes *mgmt_stypes;
  const struct ieee80211_iface_combination *iface_combinations;
  int n_iface_combinations;
  u16 software_iftypes;
  u16 n_addresses;
  u16 interface_modes;
  u16 max_acl_mac_addrs;
  u32 flags, regulatory_flags, features;
```

(continues on next page)

(continued from previous page)

```
  u8 ext_features[DIV_ROUND_UP(NUM_NL80211_EXT_FEATURES, 8)];
  u32 ap_sme_capa;
  enum cfg80211_signal_type signal_type;
  int bss_priv_size;
  u8 max_scan_ssids;
  u8 max_sched_scan_reqs;
  u8 max_sched_scan_ssids;
  u8 max_match_sets;
  u16 max_scan_ie_len;
  u16 max_sched_scan_ie_len;
  u32 max_sched_scan_plans;
  u32 max_sched_scan_plan_interval;
  u32 max_sched_scan_plan_iterations;
  int n_cipher_suites;
  const u32 *cipher_suites;
  int n_akm_suites;
  const u32 *akm_suites;
  const struct wiphy_iftype_akm_suites *iftype_akm_suites;
  unsigned int num_iftype_akm_suites;
  u8 retry_short;
  u8 retry_long;
  u32 frag_threshold;
  u32 rts_threshold;
  u8 coverage_class;
  char fw_version[ETHTOOL_FWVERS_LEN];
  u32 hw_version;
#ifdef CONFIG_PM;
  const struct wiphy_wowlan_support *wowlan;
  struct cfg80211_wowlan *wowlan_config;
#endif;
  u16 max_remain_on_channel_duration;
  u8 max_num_pmkids;
  u32 available_antennas_tx;
  u32 available_antennas_rx;
  u32 probe_resp_offload;
  const u8 *extended_capabilities, *extended_capabilities_mask;
  u8 extended_capabilities_len;
  const struct wiphy_iftype_ext_capab *iftype_ext_capab;
  unsigned int num_iftype_ext_capab;
  const void *privid;
  struct ieee80211_supported_band *bands[NUM_NL80211_BANDS];
  void (*reg_notifier)(struct wiphy *wiphy, struct regulatory_request␣
↪*request);
  const struct ieee80211_regdomain __rcu *regd;
  struct device dev;
  bool registered;
  struct dentry *debugfsdir;
  const struct ieee80211_ht_cap *ht_capa_mod_mask;
  const struct ieee80211_vht_cap *vht_capa_mod_mask;
  struct list_head wdev_list;
  possible_net_t _net;
#ifdef CONFIG_CFG80211_WEXT;
  const struct iw_handler_def *wext;
#endif;
  const struct wiphy_coalesce_support *coalesce;
  const struct wiphy_vendor_command *vendor_commands;
```

(continues on next page)

(continued from previous page)

```
  const struct nl80211_vendor_cmd_info *vendor_events;
  int n_vendor_commands, n_vendor_events;
  u16 max_ap_assoc_sta;
  u8 max_num_csa_counters;
  u32 bss_select_support;
  u8 nan_supported_bands;
  u32 txq_limit;
  u32 txq_memory_limit;
  u32 txq_quantum;
  unsigned long tx_queue_len;
  u8 support_mbssid:1, support_only_he_mbssid:1;
  const struct cfg80211_pmsr_capabilities *pmsr_capa;
  struct {
    u64 peer, vif;
    u8 max_retry;
  } tid_config_support;
  u8 max_data_retry_count;
  char priv[] ;
};
```

Members

perm_addr permanent MAC address of this device

addr_mask If the device supports multiple MAC addresses by masking, set this to a mask with variable bits set to 1, e.g. if the last four bits are variable then set it to 00-00-00-00-00-0f. The actual variable bits shall be determined by the interfaces added, with interfaces not matching the mask being rejected to be brought up.

addresses If the device has more than one address, set this pointer to a list of addresses (6 bytes each). The first one will be used by default for perm_addr. In this case, the mask should be set to all-zeroes. In this case it is assumed that the device can handle the same number of arbitrary MAC addresses.

mgmt_stypes bitmasks of frame subtypes that can be subscribed to or transmitted through nl80211, points to an array indexed by interface type

iface_combinations Valid interface combinations array, should not list single interface types.

n_iface_combinations number of entries in **iface_combinations** array.

software_iftypes bitmask of software interface types, these are not subject to any restrictions since they are purely managed in SW.

n_addresses number of addresses in **addresses**.

interface_modes bitmask of interfaces types valid for this wiphy, must be set by driver

max_acl_mac_addrs Maximum number of MAC addresses that the device supports for ACL.

flags wiphy flags, see `enum wiphy_flags`

regulatory_flags wiphy regulatory flags, see `enum ieee80211_regulatory_flags`

features features advertised to nl80211, see `enum nl80211_feature_flags`.

ext_features extended features advertised to nl80211, see `enum nl80211_ext_feature_index`.

ap_sme_capa AP SME capabilities, flags from `enum nl80211_ap_sme_features`.

signal_type signal type reported in `struct cfg80211_bss`.

bss_priv_size each BSS struct has private data allocated with it, this variable determines its size

max_scan_ssids maximum number of SSIDs the device can scan for in any given scan

max_sched_scan_reqs maximum number of scheduled scan requests that the device can run concurrently.

max_sched_scan_ssids maximum number of SSIDs the device can scan for in any given scheduled scan

max_match_sets maximum number of match sets the device can handle when performing a scheduled scan, 0 if filtering is not supported.

max_scan_ie_len maximum length of user-controlled IEs device can add to probe request frames transmitted during a scan, must not include fixed IEs like supported rates

max_sched_scan_ie_len same as max_scan_ie_len, but for scheduled scans

max_sched_scan_plans maximum number of scan plans (scan interval and number of iterations) for scheduled scan supported by the device.

max_sched_scan_plan_interval maximum interval (in seconds) for a single scan plan supported by the device.

max_sched_scan_plan_iterations maximum number of iterations for a single scan plan supported by the device.

n_cipher_suites number of supported cipher suites

cipher_suites supported cipher suites

n_akm_suites number of supported AKM suites

akm_suites supported AKM suites. These are the default AKMs supported if the supported AKMs not advertized for a specific interface type in iftype_akm_suites.

iftype_akm_suites array of supported akm suites info per interface type. Note that the bits in **iftypes_mask** inside this structure cannot overlap (i.e. only one occurrence of each type is allowed across all instances of iftype_akm_suites).

num_iftype_akm_suites number of interface types for which supported akm suites are specified separately.

retry_short Retry limit for short frames (dot11ShortRetryLimit)

retry_long Retry limit for long frames (dot11LongRetryLimit)

frag_threshold Fragmentation threshold (dot11FragmentationThreshold); -1 = fragmentation disabled, only odd values >= 256 used

rts_threshold RTS threshold (dot11RTSThreshold); -1 = RTS/CTS disabled

coverage_class current coverage class

fw_version firmware version for ethtool reporting

hw_version hardware version for ethtool reporting

wowlan WoWLAN support information

wowlan_config current WoWLAN configuration; this should usually not be used since access to it is necessarily racy, use the parameter passed to the suspend() operation instead.

max_remain_on_channel_duration Maximum time a remain-on-channel operation may request, if implemented.

max_num_pmkids maximum number of PMKIDs supported by device

available_antennas_tx bitmap of antennas which are available to be configured as TX antennas. Antenna configuration commands will be rejected unless this or **available_antennas_rx** is set.

available_antennas_rx bitmap of antennas which are available to be configured as RX antennas. Antenna configuration commands will be rejected unless this or **available_antennas_tx** is set.

probe_resp_offload Bitmap of supported protocols for probe response offloading. See `enum nl80211_probe_resp_offload_support_attr`. Only valid when the wiphy flag **WIPHY_FLAG_AP_PROBE_RESP_OFFLOAD** is set.

extended_capabilities extended capabilities supported by the driver, additional capabilities might be supported by userspace; these are the 802.11 extended capabilities (“Extended Capabilities element”) and are in the same format as in the information element. See 802.11-2012 8.4.2.29 for the defined fields. These are the default extended capabilities to be used if the capabilities are not specified for a specific interface type in iftype_ext_capab.

extended_capabilities_mask mask of the valid values

extended_capabilities_len length of the extended capabilities

iftype_ext_capab array of extended capabilities per interface type

num_iftype_ext_capab number of interface types for which extended capabilities are specified separately.

privid a pointer that drivers can use to identify if an arbitrary wiphy is theirs, e.g. in global notifiers

bands information about bands/channels supported by this device

reg_notifier the driver’s regulatory notification callback, note that if your driver uses `wiphy_apply_custom_regulatory()` the reg_notifier’s request can be passed as NULL

regd the driver’s regulatory domain, if one was requested via the `regulatory_hint()` API. This can be used by the driver on the reg_notifier()

if it chooses to ignore future regulatory domain changes caused by other drivers.

dev (virtual) struct device for this wiphy. The item in /sys/class/ieee80211/ points to this. You need use `set_wiphy_dev()` (see below).

registered protects ->resume and ->suspend sysfs callbacks against unregister hardware

debugfsdir debugfs directory used for this wiphy (ieee80211/<wiphyname>). It will be renamed automatically on wiphy renames

ht_capa_mod_mask Specify what ht_cap values can be over-ridden. If null, then none can be over-ridden.

vht_capa_mod_mask Specify what VHT capabilities can be over-ridden. If null, then none can be over-ridden.

wdev_list the list of associated (virtual) interfaces; this list must not be modified by the driver, but can be read with RTNL/RCU protection.

_net the network namespace this wiphy currently lives in

wext wireless extension handlers

coalesce packet coalescing support information

vendor_commands array of vendor commands supported by the hardware

vendor_events array of vendor events supported by the hardware

n_vendor_commands number of vendor commands

n_vendor_events number of vendor events

max_ap_assoc_sta maximum number of associated stations supported in AP mode (including P2P GO) or 0 to indicate no such limit is advertised. The driver is allowed to advertise a theoretical limit that it can reach in some cases, but may not always reach.

max_num_csa_counters Number of supported csa_counters in beacons and probe responses. This value should be set if the driver wishes to limit the number of csa counters. Default (0) means infinite.

bss_select_support bitmask indicating the BSS selection criteria supported by the driver in the .connect() callback. The bit position maps to the attribute indices defined in `enum nl80211_bss_select_attr`.

nan_supported_bands bands supported by the device in NAN mode, a bitmap of `enum nl80211_band` values. For instance, for NL80211_BAND_2GHZ, bit 0 would be set (i.e. BIT(NL80211_BAND_2GHZ)).

txq_limit configuration of internal TX queue frame limit

txq_memory_limit configuration internal TX queue memory limit

txq_quantum configuration of internal TX queue scheduler quantum

tx_queue_len allow setting transmit queue len for drivers not using wake_tx_queue

support_mbssid can HW support association with nontransmitted AP

support_only_he_mbssid don't parse MBSSID elements if it is not HE AP, in order to avoid compatibility issues. **support_mbssid** must be set for this to have any effect.

pmsr_capa peer measurement capabilities

tid_config_support describes the per-TID config support that the device has

tid_config_support.peer bitmap of attributes (configurations) supported by the driver for each peer

tid_config_support.vif bitmap of attributes (configurations) supported by the driver for each vif

tid_config_support.max_retry maximum supported retry count for long/short retry configuration

max_data_retry_count maximum supported per TID retry count for configuration through the `NL80211_TID_CONFIG_ATTR_RETRY_SHORT` and `NL80211_TID_CONFIG_ATTR_RETRY_LONG` attributes

priv driver private data (sized according to `wiphy_new()` parameter)

struct **wireless_dev**
wireless device state

Definition

```
struct wireless_dev {
  struct wiphy *wiphy;
  enum nl80211_iftype iftype;
  struct list_head list;
  struct net_device *netdev;
  u32 identifier;
  struct list_head mgmt_registrations;
  spinlock_t mgmt_registrations_lock;
  u8 mgmt_registrations_need_update:1;
  struct mutex mtx;
  bool use_4addr, is_running;
  u8 address[ETH_ALEN] ;
  u8 ssid[IEEE80211_MAX_SSID_LEN];
  u8 ssid_len, mesh_id_len, mesh_id_up_len;
  struct cfg80211_conn *conn;
  struct cfg80211_cached_keys *connect_keys;
  enum ieee80211_bss_type conn_bss_type;
  u32 conn_owner_nlportid;
  struct work_struct disconnect_wk;
  u8 disconnect_bssid[ETH_ALEN];
  struct list_head event_list;
  spinlock_t event_lock;
  struct cfg80211_internal_bss *current_bss;
  struct cfg80211_chan_def preset_chandef;
  struct cfg80211_chan_def chandef;
  bool ibss_fixed;
  bool ibss_dfs_possible;
  bool ps;
  int ps_timeout;
  int beacon_interval;
  u32 ap_unexpected_nlportid;
```

(continues on next page)

(continued from previous page)

```
  u32 owner_nlportid;
  bool nl_owner_dead;
  bool cac_started;
  unsigned long cac_start_time;
  unsigned int cac_time_ms;
#ifdef CONFIG_CFG80211_WEXT;
  struct {
    struct cfg80211_ibss_params ibss;
    struct cfg80211_connect_params connect;
    struct cfg80211_cached_keys *keys;
    const u8 *ie;
    size_t ie_len;
    u8 bssid[ETH_ALEN];
    u8 prev_bssid[ETH_ALEN];
    u8 ssid[IEEE80211_MAX_SSID_LEN];
    s8 default_key, default_mgmt_key;
    bool prev_bssid_valid;
  } wext;
#endif;
  struct cfg80211_cqm_config *cqm_config;
  struct list_head pmsr_list;
  spinlock_t pmsr_lock;
  struct work_struct pmsr_free_wk;
  unsigned long unprot_beacon_reported;
};
```

Members

wiphy pointer to hardware description

iftype interface type

list (private) Used to collect the interfaces

netdev (private) Used to reference back to the netdev, may be NULL

identifier (private) Identifier used in nl80211 to identify this wireless device if it has no netdev

mgmt_registrations list of registrations for management frames

mgmt_registrations_lock lock for the list

mgmt_registrations_need_update mgmt registrations were updated, need to propagate the update to the driver

mtx mutex used to lock data in this struct, may be used by drivers and some API functions require it held

use_4addr indicates 4addr mode is used on this interface, must be set by driver (if supported) on add_interface BEFORE registering the netdev and may otherwise be used by driver read-only, will be update by cfg80211 on change_interface

is_running true if this is a non-netdev device that has been started, e.g. the P2P Device.

address The address for this device, valid only if **netdev** is NULL

ssid (private) Used by the internal configuration code

ssid_len (private) Used by the internal configuration code

mesh_id_len (private) Used by the internal configuration code

mesh_id_up_len (private) Used by the internal configuration code

conn (private) cfg80211 software SME connection state machine data

connect_keys (private) keys to set after connection is established

conn_bss_type connecting/connected BSS type

conn_owner_nlportid (private) connection owner socket port ID

disconnect_wk (private) auto-disconnect work

disconnect_bssid (private) the BSSID to use for auto-disconnect

event_list (private) list for internal event processing

event_lock (private) lock for event list

current_bss (private) Used by the internal configuration code

preset_chandef (private) Used by the internal configuration code to track the channel to be used for AP later

chandef (private) Used by the internal configuration code to track the user-set channel definition.

ibss_fixed (private) IBSS is using fixed BSSID

ibss_dfs_possible (private) IBSS may change to a DFS channel

ps powersave mode is enabled

ps_timeout dynamic powersave timeout

beacon_interval beacon interval used on this device for transmitting beacons, 0 when not valid

ap_unexpected_nlportid (private) netlink port ID of application registered for unexpected class 3 frames (AP mode)

owner_nlportid (private) owner socket port ID

nl_owner_dead (private) owner socket went away

cac_started true if DFS channel availability check has been started

cac_start_time timestamp (jiffies) when the dfs state was entered.

cac_time_ms CAC time in ms

wext (private) Used by the internal wireless extensions compat code

wext.ibss (private) IBSS data part of wext handling

wext.connect (private) connection handling data

wext.keys (private) (WEP) key data

wext.ie (private) extra elements for association

wext.ie_len (private) length of extra elements

wext.bssid (private) selected network BSSID

wext.prev_bssid (private) previous BSSID for reassociation

wext.ssid (private) selected network SSID

wext.default_key (private) selected default key index

wext.default_mgmt_key (private) selected default management key index

wext.prev_bssid_valid (private) previous BSSID validity

cqm_config (private) nl80211 RSSI monitor state

pmsr_list (private) peer measurement requests

pmsr_lock (private) peer measurements requests/results lock

pmsr_free_wk (private) peer measurements cleanup work

unprot_beacon_reported (private) timestamp of last unprotected beacon report

Description

For netdevs, this structure must be allocated by the driver that uses the ieee80211_ptr field in struct net_device (this is intentional so it can be allocated along with the netdev.) It need not be registered then as netdev registration will be intercepted by cfg80211 to see the new wireless device.

For non-netdev uses, it must also be allocated by the driver in response to the cfg80211 callbacks that require it, as there's no netdev registration in that case it may not be allocated outside of callback operations that return it.

struct wiphy * **wiphy_new**(const struct cfg80211_ops * ops, int sizeof_priv)
create a new wiphy for use with cfg80211

Parameters

const struct cfg80211_ops * ops The configuration operations for this device

int sizeof_priv The size of the private area to allocate

Description

Create a new wiphy and associate the given operations with it. **sizeof_priv** bytes are allocated for private use.

Return

A pointer to the new wiphy. This pointer must be assigned to each netdev's ieee80211_ptr for proper operation.

void **wiphy_read_of_freq_limits**(struct wiphy * wiphy)
read frequency limits from device tree

Parameters

struct wiphy * wiphy the wireless device to get extra limits for

Description

Some devices may have extra limitations specified in DT. This may be useful for chipsets that normally support more bands but are limited due to board design (e.g. by antennas or external power amplifier).

This function reads info from DT and uses it to modify channels (disable unavailable ones). It's usually a bad idea to use it in drivers with shared channel data as DT limitations are device specific. You should make sure to call it only if channels in wiphy are copied and can be modified without affecting other devices.

As this function access device node it has to be called after set_wiphy_dev. It also modifies channels so they have to be set first. If using this helper, call it before `wiphy_register()`.

int **wiphy_register**(struct wiphy * wiphy)
register a wiphy with cfg80211

Parameters

struct wiphy * wiphy The wiphy to register.

Return

A non-negative wiphy index or a negative error code.

void **wiphy_unregister**(struct wiphy * wiphy)
deregister a wiphy from cfg80211

Parameters

struct wiphy * wiphy The wiphy to unregister.

Description

After this call, no more requests can be made with this priv pointer, but the call may sleep to wait for an outstanding request that is being handled.

void **wiphy_free**(struct wiphy * wiphy)
free wiphy

Parameters

struct wiphy * wiphy The wiphy to free

const char * **wiphy_name**(const struct wiphy * wiphy)
get wiphy name

Parameters

const struct wiphy * wiphy The wiphy whose name to return

Return

The name of **wiphy**.

struct device * **wiphy_dev**(struct wiphy * wiphy)
get wiphy dev pointer

Parameters

struct wiphy * wiphy The wiphy whose device struct to look up

Return

The dev of **wiphy**.

void * **wiphy_priv**(struct wiphy * wiphy)
return priv from wiphy

Parameters

struct wiphy * wiphy the wiphy whose priv pointer to return

Return

The priv of **wiphy**.

struct wiphy * **priv_to_wiphy**(void * priv)
return the wiphy containing the priv

Parameters

void * priv a pointer previously returned by wiphy_priv

Return

The wiphy of **priv**.

void **set_wiphy_dev**(struct wiphy * wiphy, struct device * dev)
set device pointer for wiphy

Parameters

struct wiphy * wiphy The wiphy whose device to bind

struct device * dev The device to parent it to

void * **wdev_priv**(struct wireless_dev * wdev)
return wiphy priv from wireless_dev

Parameters

struct wireless_dev * wdev The wireless device whose wiphy's priv pointer to return

Return

The wiphy priv of **wdev**.

struct **ieee80211_iface_limit**
limit on certain interface types

Definition

```
struct ieee80211_iface_limit {
  u16 max;
  u16 types;
};
```

Members

max maximum number of interfaces of these types

types interface types (bits)

struct **ieee80211_iface_combination**
possible interface combination

Definition

```
struct ieee80211_iface_combination {
  const struct ieee80211_iface_limit *limits;
  u32 num_different_channels;
  u16 max_interfaces;
  u8 n_limits;
  bool beacon_int_infra_match;
  u8 radar_detect_widths;
  u8 radar_detect_regions;
  u32 beacon_int_min_gcd;
};
```

Members

limits limits for the given interface types

num_different_channels can use up to this many different channels

max_interfaces maximum number of interfaces in total allowed in this group

n_limits number of limitations

beacon_int_infra_match In this combination, the beacon intervals between infrastructure and AP types must match. This is required only in special cases.

radar_detect_widths bitmap of channel widths supported for radar detection

radar_detect_regions bitmap of regions supported for radar detection

beacon_int_min_gcd This interface combination supports different beacon intervals.

= 0 all beacon intervals for different interface must be same.

> 0 any beacon interval for the interface part of this combination AND GCD of all beacon intervals from beaconing interfaces of this combination must be greater or equal to this value.

Description

With this structure the driver can describe which interface combinations it supports concurrently.

1. Allow #STA <= 1, #AP <= 1, matching BI, channels = 1, 2 total:

```
struct ieee80211_iface_limit limits1[] = {
        { .max = 1, .types = BIT(NL80211_IFTYPE_STATION), },
        { .max = 1, .types = BIT(NL80211_IFTYPE_AP}, },
};
struct ieee80211_iface_combination combination1 = {
        .limits = limits1,
        .n_limits = ARRAY_SIZE(limits1),
        .max_interfaces = 2,
        .beacon_int_infra_match = true,
};
```

2. Allow #{AP, P2P-GO} <= 8, channels = 1, 8 total:

```
struct ieee80211_iface_limit limits2[] = {
        { .max = 8, .types = BIT(NL80211_IFTYPE_AP) |
```

(continues on next page)

(continued from previous page)

```
                                BIT(NL80211_IFTYPE_P2P_GO), },
};
struct ieee80211_iface_combination combination2 = {
        .limits = limits2,
        .n_limits = ARRAY_SIZE(limits2),
        .max_interfaces = 8,
        .num_different_channels = 1,
};
```

3. Allow #STA <= 1, #{P2P-client,P2P-GO} <= 3 on two channels, 4 total.

   This allows for an infrastructure connection and three P2P connections.

```
struct ieee80211_iface_limit limits3[] = {
        { .max = 1, .types = BIT(NL80211_IFTYPE_STATION), },
        { .max = 3, .types = BIT(NL80211_IFTYPE_P2P_GO) |
                             BIT(NL80211_IFTYPE_P2P_CLIENT), },
};
struct ieee80211_iface_combination combination3 = {
        .limits = limits3,
        .n_limits = ARRAY_SIZE(limits3),
        .max_interfaces = 4,
        .num_different_channels = 2,
};
```

Examples

int **cfg80211_check_combinations**(struct wiphy * wiphy, struct iface_combination_params * params)
check interface combinations

Parameters

struct wiphy * wiphy the wiphy

struct iface_combination_params * params the interface combinations parameter

Description

This function can be called by the driver to check whether a combination of interfaces and their types are allowed according to the interface combinations.

47.2.2 Actions and configuration

Each wireless device and each virtual interface offer a set of configuration operations and other actions that are invoked by userspace. Each of these actions is described in the operations structure, and the parameters these operations use are described separately.

Additionally, some operations are asynchronous and expect to get status information via some functions that drivers need to call.

Scanning and BSS list handling with its associated functionality is described in a separate chapter.

struct **cfg80211_ops**
backend description for wireless configuration

Definition

```
struct cfg80211_ops {
  int (*suspend)(struct wiphy *wiphy, struct cfg80211_wowlan *wow);
  int (*resume)(struct wiphy *wiphy);
  void (*set_wakeup)(struct wiphy *wiphy, bool enabled);
  struct wireless_dev * (*add_virtual_intf)(struct wiphy *wiphy,const char␣
↪*name,unsigned char name_assign_type,enum nl80211_iftype type, struct␣
↪vif_params *params);
  int (*del_virtual_intf)(struct wiphy *wiphy, struct wireless_dev *wdev);
  int (*change_virtual_intf)(struct wiphy *wiphy,struct net_device *dev,
↪enum nl80211_iftype type, struct vif_params *params);
  int (*add_key)(struct wiphy *wiphy, struct net_device *netdev,u8 key_
↪index, bool pairwise, const u8 *mac_addr, struct key_params *params);
  int (*get_key)(struct wiphy *wiphy, struct net_device *netdev,u8 key_
↪index, bool pairwise, const u8 *mac_addr,void *cookie, void␣
↪(*callback)(void *cookie, struct key_params*));
  int (*del_key)(struct wiphy *wiphy, struct net_device *netdev, u8 key_
↪index, bool pairwise, const u8 *mac_addr);
  int (*set_default_key)(struct wiphy *wiphy,struct net_device *netdev, u8␣
↪key_index, bool unicast, bool multicast);
  int (*set_default_mgmt_key)(struct wiphy *wiphy,struct net_device␣
↪*netdev, u8 key_index);
  int (*set_default_beacon_key)(struct wiphy *wiphy,struct net_device␣
↪*netdev, u8 key_index);
  int (*start_ap)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_ap_settings *settings);
  int (*change_beacon)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_beacon_data *info);
  int (*stop_ap)(struct wiphy *wiphy, struct net_device *dev);
  int (*add_station)(struct wiphy *wiphy, struct net_device *dev,const u8␣
↪*mac, struct station_parameters *params);
  int (*del_station)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪station_del_parameters *params);
  int (*change_station)(struct wiphy *wiphy, struct net_device *dev,const␣
↪u8 *mac, struct station_parameters *params);
  int (*get_station)(struct wiphy *wiphy, struct net_device *dev, const u8␣
↪*mac, struct station_info *sinfo);
  int (*dump_station)(struct wiphy *wiphy, struct net_device *dev, int idx,
↪ u8 *mac, struct station_info *sinfo);
  int (*add_mpath)(struct wiphy *wiphy, struct net_device *dev, const u8␣
↪*dst, const u8 *next_hop);
  int (*del_mpath)(struct wiphy *wiphy, struct net_device *dev, const u8␣
↪*dst);
  int (*change_mpath)(struct wiphy *wiphy, struct net_device *dev, const␣
↪u8 *dst, const u8 *next_hop);
  int (*get_mpath)(struct wiphy *wiphy, struct net_device *dev, u8 *dst,␣
↪u8 *next_hop, struct mpath_info *pinfo);
  int (*dump_mpath)(struct wiphy *wiphy, struct net_device *dev,int idx,␣
↪u8 *dst, u8 *next_hop, struct mpath_info *pinfo);
  int (*get_mpp)(struct wiphy *wiphy, struct net_device *dev, u8 *dst, u8␣
↪*mpp, struct mpath_info *pinfo);
  int (*dump_mpp)(struct wiphy *wiphy, struct net_device *dev,int idx, u8␣
↪*dst, u8 *mpp, struct mpath_info *pinfo);
```

(continues on next page)

(continued from previous page)

```
  int (*get_mesh_config)(struct wiphy *wiphy,struct net_device *dev,␣
↪struct mesh_config *conf);
  int (*update_mesh_config)(struct wiphy *wiphy,struct net_device *dev,␣
↪u32 mask, const struct mesh_config *nconf);
  int (*join_mesh)(struct wiphy *wiphy, struct net_device *dev,const␣
↪struct mesh_config *conf, const struct mesh_setup *setup);
  int (*leave_mesh)(struct wiphy *wiphy, struct net_device *dev);
  int (*join_ocb)(struct wiphy *wiphy, struct net_device *dev, struct ocb_
↪setup *setup);
  int (*leave_ocb)(struct wiphy *wiphy, struct net_device *dev);
  int (*change_bss)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪bss_parameters *params);
  int (*set_txq_params)(struct wiphy *wiphy, struct net_device *dev,␣
↪struct ieee80211_txq_params *params);
  int (*libertas_set_mesh_channel)(struct wiphy *wiphy,struct net_device␣
↪*dev, struct ieee80211_channel *chan);
  int (*set_monitor_channel)(struct wiphy *wiphy, struct cfg80211_chan_def␣
↪*chandef);
  int (*scan)(struct wiphy *wiphy, struct cfg80211_scan_request *request);
  void (*abort_scan)(struct wiphy *wiphy, struct wireless_dev *wdev);
  int (*auth)(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_
↪auth_request *req);
  int (*assoc)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_assoc_request *req);
  int (*deauth)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_deauth_request *req);
  int (*disassoc)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_disassoc_request *req);
  int (*connect)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_connect_params *sme);
  int (*update_connect_params)(struct wiphy *wiphy,struct net_device *dev,
↪struct cfg80211_connect_params *sme, u32 changed);
  int (*disconnect)(struct wiphy *wiphy, struct net_device *dev, u16␣
↪reason_code);
  int (*join_ibss)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_ibss_params *params);
  int (*leave_ibss)(struct wiphy *wiphy, struct net_device *dev);
  int (*set_mcast_rate)(struct wiphy *wiphy, struct net_device *dev, int␣
↪rate[NUM_NL80211_BANDS]);
  int (*set_wiphy_params)(struct wiphy *wiphy, u32 changed);
  int (*set_tx_power)(struct wiphy *wiphy, struct wireless_dev *wdev, enum␣
↪nl80211_tx_power_setting type, int mbm);
  int (*get_tx_power)(struct wiphy *wiphy, struct wireless_dev *wdev, int␣
↪*dbm);
  int (*set_wds_peer)(struct wiphy *wiphy, struct net_device *dev, const␣
↪u8 *addr);
  void (*rfkill_poll)(struct wiphy *wiphy);
#ifdef CONFIG_NL80211_TESTMODE;
  int (*testmode_cmd)(struct wiphy *wiphy, struct wireless_dev *wdev, void␣
↪*data, int len);
  int (*testmode_dump)(struct wiphy *wiphy, struct sk_buff *skb,struct␣
↪netlink_callback *cb, void *data, int len);
#endif;
  int (*set_bitrate_mask)(struct wiphy *wiphy,struct net_device *dev,const␣
↪u8 *peer, const struct cfg80211_bitrate_mask *mask);
  int (*dump_survey)(struct wiphy *wiphy, struct net_device *netdev, int␣
↪idx, struct survey_info *info);
```

(continues on next page)

(continued from previous page)

```
 int (*set_pmksa)(struct wiphy *wiphy, struct net_device *netdev, struct␣
↪cfg80211_pmksa *pmksa);
 int (*del_pmksa)(struct wiphy *wiphy, struct net_device *netdev, struct␣
↪cfg80211_pmksa *pmksa);
 int (*flush_pmksa)(struct wiphy *wiphy, struct net_device *netdev);
 int (*remain_on_channel)(struct wiphy *wiphy,struct wireless_dev *wdev,
↪struct ieee80211_channel *chan,unsigned int duration, u64 *cookie);
 int (*cancel_remain_on_channel)(struct wiphy *wiphy,struct wireless_dev␣
↪*wdev, u64 cookie);
 int (*mgmt_tx)(struct wiphy *wiphy, struct wireless_dev *wdev,struct␣
↪cfg80211_mgmt_tx_params *params, u64 *cookie);
 int (*mgmt_tx_cancel_wait)(struct wiphy *wiphy,struct wireless_dev *wdev,
↪ u64 cookie);
 int (*set_power_mgmt)(struct wiphy *wiphy, struct net_device *dev, bool␣
↪enabled, int timeout);
 int (*set_cqm_rssi_config)(struct wiphy *wiphy,struct net_device *dev,␣
↪s32 rssi_thold, u32 rssi_hyst);
 int (*set_cqm_rssi_range_config)(struct wiphy *wiphy,struct net_device␣
↪*dev, s32 rssi_low, s32 rssi_high);
 int (*set_cqm_txe_config)(struct wiphy *wiphy,struct net_device *dev,␣
↪u32 rate, u32 pkts, u32 intvl);
 void (*update_mgmt_frame_registrations)(struct wiphy *wiphy,struct␣
↪wireless_dev *wdev, struct mgmt_frame_regs *upd);
 int (*set_antenna)(struct wiphy *wiphy, u32 tx_ant, u32 rx_ant);
 int (*get_antenna)(struct wiphy *wiphy, u32 *tx_ant, u32 *rx_ant);
 int (*sched_scan_start)(struct wiphy *wiphy,struct net_device *dev,␣
↪struct cfg80211_sched_scan_request *request);
 int (*sched_scan_stop)(struct wiphy *wiphy, struct net_device *dev, u64␣
↪reqid);
 int (*set_rekey_data)(struct wiphy *wiphy, struct net_device *dev,␣
↪struct cfg80211_gtk_rekey_data *data);
 int (*tdls_mgmt)(struct wiphy *wiphy, struct net_device *dev,const u8␣
↪*peer, u8 action_code,  u8 dialog_token,u16 status_code, u32 peer_
↪capability, bool initiator, const u8 *buf, size_t len);
 int (*tdls_oper)(struct wiphy *wiphy, struct net_device *dev, const u8␣
↪*peer, enum nl80211_tdls_operation oper);
 int (*probe_client)(struct wiphy *wiphy, struct net_device *dev, const␣
↪u8 *peer, u64 *cookie);
 int (*set_noack_map)(struct wiphy *wiphy,struct net_device *dev, u16␣
↪noack_map);
 int (*get_channel)(struct wiphy *wiphy,struct wireless_dev *wdev, struct␣
↪cfg80211_chan_def *chandef);
 int (*start_p2p_device)(struct wiphy *wiphy, struct wireless_dev *wdev);
 void (*stop_p2p_device)(struct wiphy *wiphy, struct wireless_dev *wdev);
 int (*set_mac_acl)(struct wiphy *wiphy, struct net_device *dev, const␣
↪struct cfg80211_acl_data *params);
 int (*start_radar_detection)(struct wiphy *wiphy,struct net_device *dev,
↪struct cfg80211_chan_def *chandef, u32 cac_time_ms);
 void (*end_cac)(struct wiphy *wiphy, struct net_device *dev);
 int (*update_ft_ies)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_update_ft_ies_params *ftie);
 int (*crit_proto_start)(struct wiphy *wiphy,struct wireless_dev *wdev,
↪enum nl80211_crit_proto_id protocol, u16 duration);
 void (*crit_proto_stop)(struct wiphy *wiphy, struct wireless_dev *wdev);
 int (*set_coalesce)(struct wiphy *wiphy, struct cfg80211_coalesce␣
↪*coalesce);
```

(continues on next page)

(continued from previous page)

```
  int (*channel_switch)(struct wiphy *wiphy,struct net_device *dev, struct␣
↪cfg80211_csa_settings *params);
  int (*set_qos_map)(struct wiphy *wiphy,struct net_device *dev, struct␣
↪cfg80211_qos_map *qos_map);
  int (*set_ap_chanwidth)(struct wiphy *wiphy, struct net_device *dev,␣
↪struct cfg80211_chan_def *chandef);
  int (*add_tx_ts)(struct wiphy *wiphy, struct net_device *dev,u8 tsid,␣
↪const u8 *peer, u8 user_prio, u16 admitted_time);
  int (*del_tx_ts)(struct wiphy *wiphy, struct net_device *dev, u8 tsid,␣
↪const u8 *peer);
  int (*tdls_channel_switch)(struct wiphy *wiphy,struct net_device *dev,
↪const u8 *addr, u8 oper_class, struct cfg80211_chan_def *chandef);
  void (*tdls_cancel_channel_switch)(struct wiphy *wiphy,struct net_device␣
↪*dev, const u8 *addr);
  int (*start_nan)(struct wiphy *wiphy, struct wireless_dev *wdev, struct␣
↪cfg80211_nan_conf *conf);
  void (*stop_nan)(struct wiphy *wiphy, struct wireless_dev *wdev);
  int (*add_nan_func)(struct wiphy *wiphy, struct wireless_dev *wdev,␣
↪struct cfg80211_nan_func *nan_func);
  void (*del_nan_func)(struct wiphy *wiphy, struct wireless_dev *wdev, u64␣
↪cookie);
  int (*nan_change_conf)(struct wiphy *wiphy,struct wireless_dev *wdev,
↪struct cfg80211_nan_conf *conf, u32 changes);
  int (*set_multicast_to_unicast)(struct wiphy *wiphy,struct net_device␣
↪*dev, const bool enabled);
  int (*get_txq_stats)(struct wiphy *wiphy,struct wireless_dev *wdev,␣
↪struct cfg80211_txq_stats *txqstats);
  int (*set_pmk)(struct wiphy *wiphy, struct net_device *dev, const struct␣
↪cfg80211_pmk_conf *conf);
  int (*del_pmk)(struct wiphy *wiphy, struct net_device *dev, const u8␣
↪*aa);
  int (*external_auth)(struct wiphy *wiphy, struct net_device *dev, struct␣
↪cfg80211_external_auth_params *params);
  int (*tx_control_port)(struct wiphy *wiphy,struct net_device *dev,const␣
↪u8 *buf, size_t len,const u8 *dest, const __be16 proto,const bool␣
↪noencrypt, u64 *cookie);
  int (*get_ftm_responder_stats)(struct wiphy *wiphy,struct net_device␣
↪*dev, struct cfg80211_ftm_responder_stats *ftm_stats);
  int (*start_pmsr)(struct wiphy *wiphy, struct wireless_dev *wdev, struct␣
↪cfg80211_pmsr_request *request);
  void (*abort_pmsr)(struct wiphy *wiphy, struct wireless_dev *wdev,␣
↪struct cfg80211_pmsr_request *request);
  int (*update_owe_info)(struct wiphy *wiphy, struct net_device *dev,␣
↪struct cfg80211_update_owe_info *owe_info);
  int (*probe_mesh_link)(struct wiphy *wiphy, struct net_device *dev,␣
↪const u8 *buf, size_t len);
  int (*set_tid_config)(struct wiphy *wiphy, struct net_device *dev,␣
↪struct cfg80211_tid_config *tid_conf);
  int (*reset_tid_config)(struct wiphy *wiphy, struct net_device *dev,␣
↪const u8 *peer, u8 tids);
};
```

Members

suspend wiphy device needs to be suspended. The variable **wow** will be NULL or contain the enabled Wake-on-Wireless triggers that are configured for the

device.

resume wiphy device needs to be resumed

set_wakeup Called when WoWLAN is enabled/disabled, use this callback to call device_set_wakeup_enable() to enable/disable wakeup from the device.

add_virtual_intf create a new virtual interface with the given name, must set the struct wireless_dev's iftype. Beware: You must create the new netdev in the wiphy's network namespace! Returns the struct wireless_dev, or an ERR_PTR. For P2P device wdevs, the driver must also set the address member in the wdev.

del_virtual_intf remove the virtual interface

change_virtual_intf change type/configuration of virtual interface, keep the struct wireless_dev's iftype updated.

add_key add a key with the given parameters. **mac_addr** will be NULL when adding a group key.

get_key get information about the key with the given parameters. **mac_addr** will be NULL when requesting information for a group key. All pointers given to the **callback** function need not be valid after it returns. This function should return an error if it is not possible to retrieve the key, -ENOENT if it doesn't exist.

del_key remove a key given the **mac_addr** (NULL for a group key) and **key_index**, return -ENOENT if the key doesn't exist.

set_default_key set the default key on an interface

set_default_mgmt_key set the default management frame key on an interface

set_default_beacon_key set the default Beacon frame key on an interface

start_ap Start acting in AP mode defined by the parameters.

change_beacon Change the beacon parameters for an access point mode interface. This should reject the call when AP mode wasn't started.

stop_ap Stop being an AP, including stopping beaconing.

add_station Add a new station.

del_station Remove a station

change_station Modify a given station. Note that flags changes are not much validated in cfg80211, in particular the auth/assoc/authorized flags might come to the driver in invalid combinations – make sure to check them, also against the existing state! Drivers must call cfg80211_check_station_change() to validate the information.

get_station get station information for the station identified by **mac**

dump_station dump station callback – resume dump at index **idx**

add_mpath add a fixed mesh path

del_mpath delete a given mesh path

change_mpath change a given mesh path

get_mpath get a mesh path for the given parameters

dump_mpath dump mesh path callback – resume dump at index **idx**

get_mpp get a mesh proxy path for the given parameters

dump_mpp dump mesh proxy path callback – resume dump at index **idx**

get_mesh_config Get the current mesh configuration

update_mesh_config Update mesh parameters on a running mesh. The mask is a bitfield which tells us which parameters to set, and which to leave alone.

join_mesh join the mesh network with the specified parameters (invoked with the wireless_dev mutex held)

leave_mesh leave the current mesh network (invoked with the wireless_dev mutex held)

join_ocb join the OCB network with the specified parameters (invoked with the wireless_dev mutex held)

leave_ocb leave the current OCB network (invoked with the wireless_dev mutex held)

change_bss Modify parameters for a given BSS.

set_txq_params Set TX queue parameters

libertas_set_mesh_channel Only for backward compatibility for libertas, as it doesn't implement join_mesh and needs to set the channel to join the mesh instead.

set_monitor_channel Set the monitor mode channel for the device. If other interfaces are active this callback should reject the configuration. If no interfaces are active or the device is down, the channel should be stored for when a monitor interface becomes active.

scan Request to do a scan. If returning zero, the scan request is given the driver, and will be valid until passed to `cfg80211_scan_done()`. For scan results, call `cfg80211_inform_bss()`; you can call this outside the scan/scan_done bracket too.

abort_scan Tell the driver to abort an ongoing scan. The driver shall indicate the status of the scan through `cfg80211_scan_done()`.

auth Request to authenticate with the specified peer (invoked with the wireless_dev mutex held)

assoc Request to (re)associate with the specified peer (invoked with the wireless_dev mutex held)

deauth Request to deauthenticate from the specified peer (invoked with the wireless_dev mutex held)

disassoc Request to disassociate from the specified peer (invoked with the wireless_dev mutex held)

connect Connect to the ESS with the specified parameters. When connected, call `cfg80211_connect_result()/cfg80211_connect_bss()` with status code `WLAN_STATUS_SUCCESS`. If the connection fails for some reason,

call `cfg80211_connect_result()`/`cfg80211_connect_bss()` with the status code from the AP or `cfg80211_connect_timeout()` if no frame with status code was received. The driver is allowed to roam to other BSSes within the ESS when the other BSS matches the connect parameters. When such roaming is initiated by the driver, the driver is expected to verify that the target matches the configured security parameters and to use Reassociation Request frame instead of Association Request frame. The connect function can also be used to request the driver to perform a specific roam when connected to an ESS. In that case, the prev_bssid parameter is set to the BSSID of the currently associated BSS as an indication of requesting reassociation. In both the driver-initiated and new connect() call initiated roaming cases, the result of roaming is indicated with a call to `cfg80211_roamed()`. (invoked with the wireless_dev mutex held)

update_connect_params Update the connect parameters while connected to a BSS. The updated parameters can be used by driver/firmware for subsequent BSS selection (roaming) decisions and to form the Authentication/(Re)Association Request frames. This call does not request an immediate disassociation or reassociation with the current BSS, i.e., this impacts only subsequent (re)associations. The bits in changed are defined in `enum cfg80211_connect_params_changed`. (invoked with the wireless_dev mutex held)

disconnect Disconnect from the BSS/ESS or stop connection attempts if connection is in progress. Once done, call `cfg80211_disconnected()` in case connection was already established (invoked with the wireless_dev mutex held), otherwise call `cfg80211_connect_timeout()`.

join_ibss Join the specified IBSS (or create if necessary). Once done, call `cfg80211_ibss_joined()`, also call that function when changing BSSID due to a merge. (invoked with the wireless_dev mutex held)

leave_ibss Leave the IBSS. (invoked with the wireless_dev mutex held)

set_mcast_rate Set the specified multicast rate (only if vif is in ADHOC or MESH mode)

set_wiphy_params Notify that wiphy parameters have changed; **changed** bitfield (see `enum wiphy_params_flags`) describes which values have changed. The actual parameter values are available in struct wiphy. If returning an error, no value should be changed.

set_tx_power set the transmit power according to the parameters, the power passed is in mBm, to get dBm use MBM_TO_DBM(). The wdev may be `NULL` if power was set for the wiphy, and will always be `NULL` unless the driver supports per-vif TX power (as advertised by the nl80211 feature flag.)

get_tx_power store the current TX power into the dbm variable; return 0 if successful

set_wds_peer set the WDS peer for a WDS interface

rfkill_poll polls the hw rfkill line, use cfg80211 reporting functions to adjust rfkill hw state

testmode_cmd run a test mode command; **wdev** may be `NULL`

testmode_dump Implement a test mode dump. The cb->args[2] and up may be used by the function, but 0 and 1 must not be touched. Additionally, return error codes other than -ENOBUFS and -ENOENT will terminate the dump and return to userspace with an error, so be careful. If any data was passed in from userspace then the data/len arguments will be present and point to the data contained in `NL80211_ATTR_TESTDATA`.

set_bitrate_mask set the bitrate mask configuration

dump_survey get site survey information.

set_pmksa Cache a PMKID for a BSSID. This is mostly useful for fullmac devices running firmwares capable of generating the (re) association RSN IE. It allows for faster roaming between WPA2 BSSIDs.

del_pmksa Delete a cached PMKID.

flush_pmksa Flush all cached PMKIDs.

remain_on_channel Request the driver to remain awake on the specified channel for the specified duration to complete an off-channel operation (e.g., public action frame exchange). When the driver is ready on the requested channel, it must indicate this with an event notification by calling `cfg80211_ready_on_channel()`.

cancel_remain_on_channel Cancel an on-going remain-on-channel operation. This allows the operation to be terminated prior to timeout based on the duration value.

mgmt_tx Transmit a management frame.

mgmt_tx_cancel_wait Cancel the wait time from transmitting a management frame on another channel

set_power_mgmt Configure WLAN power management. A timeout value of -1 allows the driver to adjust the dynamic ps timeout value.

set_cqm_rssi_config Configure connection quality monitor RSSI threshold. After configuration, the driver should (soon) send an event indicating the current level is above/below the configured threshold; this may need some care when the configuration is changed (without first being disabled.)

set_cqm_rssi_range_config Configure two RSSI thresholds in the connection quality monitor. An event is to be sent only when the signal level is found to be outside the two values. The driver should set `NL80211_EXT_FEATURE_CQM_RSSI_LIST` if this method is implemented. If it is provided then there's no point providing **set_cqm_rssi_config**.

set_cqm_txe_config Configure connection quality monitor TX error thresholds.

update_mgmt_frame_registrations Notify the driver that management frame registrations were updated. The callback is allowed to sleep.

set_antenna Set antenna configuration (tx_ant, rx_ant) on the device. Parameters are bitmaps of allowed antennas to use for TX/RX. Drivers may reject TX/RX mask combinations they cannot support by returning -EINVAL (also see nl80211.h **NL80211_ATTR_WIPHY_ANTENNA_TX**).

get_antenna Get current antenna configuration from device (tx_ant, rx_ant).

sched_scan_start Tell the driver to start a scheduled scan.

sched_scan_stop Tell the driver to stop an ongoing scheduled scan with given request id. This call must stop the scheduled scan and be ready for starting a new one before it returns, i.e. **sched_scan_start** may be called immediately after that again and should not fail in that case. The driver should not call cfg80211_sched_scan_stopped() for a requested stop (when this method returns 0).

set_rekey_data give the data necessary for GTK rekeying to the driver

tdls_mgmt Transmit a TDLS management frame.

tdls_oper Perform a high-level TDLS operation (e.g. TDLS link setup).

probe_client probe an associated client, must return a cookie that it later passes to cfg80211_probe_status().

set_noack_map Set the NoAck Map for the TIDs.

get_channel Get the current operating channel for the virtual interface. For monitor interfaces, it should return `NULL` unless there's a single current monitoring channel.

start_p2p_device Start the given P2P device.

stop_p2p_device Stop the given P2P device.

set_mac_acl Sets MAC address control list in AP and P2P GO mode. Parameters include ACL policy, an array of MAC address of stations and the number of MAC addresses. If there is already a list in driver this new list replaces the existing one. Driver has to clear its ACL when number of MAC addresses entries is passed as 0. Drivers which advertise the support for MAC based ACL have to implement this callback.

start_radar_detection Start radar detection in the driver.

end_cac End running CAC, probably because a related CAC was finished on another phy.

update_ft_ies Provide updated Fast BSS Transition information to the driver. If the SME is in the driver/firmware, this information can be used in building Authentication and Reassociation Request frames.

crit_proto_start Indicates a critical protocol needs more link reliability for a given duration (milliseconds). The protocol is provided so the driver can take the most appropriate actions.

crit_proto_stop Indicates critical protocol no longer needs increased link reliability. This operation can not fail.

set_coalesce Set coalesce parameters.

channel_switch initiate channel-switch procedure (with CSA). Driver is responsible for veryfing if the switch is possible. Since this is inherently tricky driver may decide to disconnect an interface later with cfg80211_stop_iface(). This doesn't mean driver can accept everything. It should do it's best to verify requests and reject them as soon as possible.

set_qos_map Set QoS mapping information to the driver

set_ap_chanwidth Set the AP (including P2P GO) mode channel width for the given interface This is used e.g. for dynamic HT 20/40 MHz channel width changes during the lifetime of the BSS.

add_tx_ts validate (if admitted_time is 0) or add a TX TS to the device with the given parameters; action frame exchange has been handled by userspace so this just has to modify the TX path to take the TS into account. If the admitted time is 0 just validate the parameters to make sure the session can be created at all; it is valid to just always return success for that but that may result in inefficient behaviour (handshake with the peer followed by immediate teardown when the addition is later rejected)

del_tx_ts remove an existing TX TS

tdls_channel_switch Start channel-switching with a TDLS peer. The driver is responsible for continually initiating channel-switching operations and returning to the base channel for communication with the AP.

tdls_cancel_channel_switch Stop channel-switching with a TDLS peer. Both peers must be on the base channel when the call completes.

start_nan Start the NAN interface.

stop_nan Stop the NAN interface.

add_nan_func Add a NAN function. Returns negative value on failure. On success **nan_func** ownership is transferred to the driver and it may access it outside of the scope of this function. The driver should free the **nan_func** when no longer needed by calling cfg80211_free_nan_func(). On success the driver should assign an instance_id in the provided **nan_func**.

del_nan_func Delete a NAN function.

nan_change_conf changes NAN configuration. The changed parameters must be specified in **changes** (using `enum cfg80211_nan_conf_changes`); All other parameters must be ignored.

set_multicast_to_unicast configure multicast to unicast conversion for BSS

get_txq_stats Get TXQ stats for interface or phy. If wdev is `NULL`, this function should return phy stats, and interface stats otherwise.

set_pmk configure the PMK to be used for offloaded 802.1X 4-Way handshake. If not deleted through **del_pmk** the PMK remains valid until disconnect upon which the driver should clear it. (invoked with the wireless_dev mutex held)

del_pmk delete the previously configured PMK for the given authenticator. (invoked with the wireless_dev mutex held)

external_auth indicates result of offloaded authentication processing from user space

tx_control_port TX a control port frame (EAPoL). The noencrypt parameter tells the driver that the frame should not be encrypted.

get_ftm_responder_stats Retrieve FTM responder statistics, if available. Statistics should be cumulative, currently no way to reset is provided.

start_pmsr start peer measurement (e.g. FTM)

abort_pmsr abort peer measurement

update_owe_info Provide updated OWE info to driver. Driver implementing SME but offloading OWE processing to the user space will get the updated DH IE through this interface.

probe_mesh_link Probe direct Mesh peer' s link quality by sending data frame and overrule HWMP path selection algorithm.

set_tid_config TID specific configuration, this can be peer or BSS specific This callback may sleep.

reset_tid_config Reset TID specific configuration for the peer, for the given TIDs. This callback may sleep.

Description

This struct is registered by fullmac card drivers and/or wireless stacks in order to handle configuration requests on their interfaces.

All callbacks except where otherwise noted should return 0 on success or a negative error code.

All operations are currently invoked under rtnl for consistency with the wireless extensions but this is subject to reevaluation as soon as this code is used more widely and we have a first user without wext.

struct **vif_params**
describes virtual interface parameters

Definition

```
struct vif_params {
  u32 flags;
  int use_4addr;
  u8 macaddr[ETH_ALEN];
  const u8 *vht_mumimo_groups;
  const u8 *vht_mumimo_follow_addr;
};
```

Members

flags monitor interface flags, unchanged if 0, otherwise `MONITOR_FLAG_CHANGED` will be set

use_4addr use 4-address frames

macaddr address to use for this virtual interface. If this parameter is set to zero address the driver may determine the address as needed. This feature is only fully supported by drivers that enable the `NL80211_FEATURE_MAC_ON_CREATE` flag. Others may support creating * only p2p devices with specified MAC.

vht_mumimo_groups MU-MIMO groupID, used for monitoring MU-MIMO packets belonging to that MU-MIMO groupID; `NULL` if not changed

vht_mumimo_follow_addr MU-MIMO follow address, used for monitoring MU-MIMO packets going to the specified station; `NULL` if not changed

struct **key_params**
key information

Definition

```
struct key_params {
  const u8 *key;
  const u8 *seq;
  int key_len;
  int seq_len;
  u16 vlan_id;
  u32 cipher;
  enum nl80211_key_mode mode;
};
```

Members

key key material

seq sequence counter (IV/PN) for TKIP and CCMP keys, only used with the get_key() callback, must be in little endian, length given by **seq_len**.

key_len length of key material

seq_len length of **seq**.

vlan_id vlan_id for VLAN group key (if nonzero)

cipher cipher suite selector

mode key install mode (RX_TX, NO_TX or SET_TX)

Description

Information about a key

enum **survey_info_flags**
survey information flags

Constants

SURVEY_INFO_NOISE_DBM noise (in dBm) was filled in

SURVEY_INFO_IN_USE channel is currently being used

SURVEY_INFO_TIME active time (in ms) was filled in

SURVEY_INFO_TIME_BUSY busy time was filled in

SURVEY_INFO_TIME_EXT_BUSY extension channel busy time was filled in

SURVEY_INFO_TIME_RX receive time was filled in

SURVEY_INFO_TIME_TX transmit time was filled in

SURVEY_INFO_TIME_SCAN scan time was filled in

SURVEY_INFO_TIME_BSS_RX local BSS receive time was filled in

Description

Used by the driver to indicate which info in `struct survey_info` it has filled in during the get_survey().

struct **survey_info**
channel survey response

Definition

```
struct survey_info {
  struct ieee80211_channel *channel;
  u64 time;
  u64 time_busy;
  u64 time_ext_busy;
  u64 time_rx;
  u64 time_tx;
  u64 time_scan;
  u64 time_bss_rx;
  u32 filled;
  s8 noise;
};
```

Members

channel the channel this survey record reports, may be NULL for a single record to report global statistics

time amount of time in ms the radio was turn on (on the channel)

time_busy amount of time the primary channel was sensed busy

time_ext_busy amount of time the extension channel was sensed busy

time_rx amount of time the radio spent receiving data

time_tx amount of time the radio spent transmitting data

time_scan amount of time the radio spent for scanning

time_bss_rx amount of time the radio spent receiving data on a local BSS

filled bitflag of flags from enum survey_info_flags

noise channel noise in dBm. This and all following fields are optional

Description

Used by dump_survey() to report back per-channel survey information.

This structure can later be expanded with things like channel duty cycle etc.

struct **cfg80211_beacon_data**
beacon data

Definition

```
struct cfg80211_beacon_data {
  const u8 *head, *tail;
  const u8 *beacon_ies;
  const u8 *proberesp_ies;
  const u8 *assocresp_ies;
  const u8 *probe_resp;
  const u8 *lci;
  const u8 *civicloc;
  s8 ftm_responder;
  size_t head_len, tail_len;
  size_t beacon_ies_len;
  size_t proberesp_ies_len;
  size_t assocresp_ies_len;
  size_t probe_resp_len;
```

(continues on next page)

(continued from previous page)

```
  size_t lci_len;
  size_t civicloc_len;
};
```

Members

head head portion of beacon (before TIM IE) or NULL if not changed

tail tail portion of beacon (after TIM IE) or NULL if not changed

beacon_ies extra information element(s) to add into Beacon frames or NULL

proberesp_ies extra information element(s) to add into Probe Response frames or NULL

assocresp_ies extra information element(s) to add into (Re)Association Response frames or NULL

probe_resp probe response template (AP mode only)

lci Measurement Report element content, starting with Measurement Token (measurement type 8)

civicloc Measurement Report element content, starting with Measurement Token (measurement type 11)

ftm_responder enable FTM responder functionality; -1 for no change (which also implies no change in LCI/civic location data)

head_len length of **head**

tail_len length of **tail**

beacon_ies_len length of beacon_ies in octets

proberesp_ies_len length of proberesp_ies in octets

assocresp_ies_len length of assocresp_ies in octets

probe_resp_len length of probe response template (**probe_resp**)

lci_len LCI data length

civicloc_len Civic location data length

struct **cfg80211_ap_settings**
AP configuration

Definition

```
struct cfg80211_ap_settings {
  struct cfg80211_chan_def chandef;
  struct cfg80211_beacon_data beacon;
  int beacon_interval, dtim_period;
  const u8 *ssid;
  size_t ssid_len;
  enum nl80211_hidden_ssid hidden_ssid;
  struct cfg80211_crypto_settings crypto;
  bool privacy;
  enum nl80211_auth_type auth_type;
```

(continues on next page)

(continued from previous page)

```
  enum nl80211_smps_mode smps_mode;
  int inactivity_timeout;
  u8 p2p_ctwindow;
  bool p2p_opp_ps;
  const struct cfg80211_acl_data *acl;
  bool pbss;
  struct cfg80211_bitrate_mask beacon_rate;
  const struct ieee80211_ht_cap *ht_cap;
  const struct ieee80211_vht_cap *vht_cap;
  const struct ieee80211_he_cap_elem *he_cap;
  const struct ieee80211_he_operation *he_oper;
  bool ht_required, vht_required, he_required;
  bool twt_responder;
  u32 flags;
  struct ieee80211_he_obss_pd he_obss_pd;
  struct cfg80211_he_bss_color he_bss_color;
};
```

Members

chandef defines the channel to use

beacon beacon data

beacon_interval beacon interval

dtim_period DTIM period

ssid SSID to be used in the BSS (note: may be NULL if not provided from user space)

ssid_len length of **ssid**

hidden_ssid whether to hide the SSID in Beacon/Probe Response frames

crypto crypto settings

privacy the BSS uses privacy

auth_type Authentication type (algorithm)

smps_mode SMPS mode

inactivity_timeout time in seconds to determine station's inactivity.

p2p_ctwindow P2P CT Window

p2p_opp_ps P2P opportunistic PS

acl ACL configuration used by the drivers which has support for MAC address based access control

pbss If set, start as a PCP instead of AP. Relevant for DMG networks.

beacon_rate bitrate to be used for beacons

ht_cap HT capabilities (or NULL if HT isn't enabled)

vht_cap VHT capabilities (or NULL if VHT isn't enabled)

he_cap HE capabilities (or NULL if HE isn't enabled)

he_oper HE operation IE (or NULL if HE isn't enabled)

ht_required stations must support HT

vht_required stations must support VHT

he_required stations must support HE

twt_responder Enable Target Wait Time

flags flags, as defined in enum cfg80211_ap_settings_flags

he_obss_pd OBSS Packet Detection settings

he_bss_color BSS Color settings

Description

Used to configure an AP interface.

struct **station_parameters**
station parameters

Definition

```
struct station_parameters {
  const u8 *supported_rates;
  struct net_device *vlan;
  u32 sta_flags_mask, sta_flags_set;
  u32 sta_modify_mask;
  int listen_interval;
  u16 aid;
  u16 vlan_id;
  u16 peer_aid;
  u8 supported_rates_len;
  u8 plink_action;
  u8 plink_state;
  const struct ieee80211_ht_cap *ht_capa;
  const struct ieee80211_vht_cap *vht_capa;
  u8 uapsd_queues;
  u8 max_sp;
  enum nl80211_mesh_power_mode local_pm;
  u16 capability;
  const u8 *ext_capab;
  u8 ext_capab_len;
  const u8 *supported_channels;
  u8 supported_channels_len;
  const u8 *supported_oper_classes;
  u8 supported_oper_classes_len;
  u8 opmode_notif;
  bool opmode_notif_used;
  int support_p2p_ps;
  const struct ieee80211_he_cap_elem *he_capa;
  u8 he_capa_len;
  u16 airtime_weight;
  struct sta_txpwr txpwr;
  const struct ieee80211_he_6ghz_capa *he_6ghz_capa;
};
```

Members

supported_rates supported rates in IEEE 802.11 format (or NULL for no change)

vlan vlan interface station should belong to

sta_flags_mask station flags that changed (bitmask of BIT(NL80211_STA_FLAG_…))

sta_flags_set station flags values (bitmask of BIT(NL80211_STA_FLAG_…))

sta_modify_mask bitmap indicating which parameters changed (for those that don't have a natural "no change" value), see enum station_parameters_apply_mask

listen_interval listen interval or -1 for no change

aid AID or zero for no change

vlan_id VLAN ID for station (if nonzero)

peer_aid mesh peer AID or zero for no change

supported_rates_len number of supported rates

plink_action plink action to take

plink_state set the peer link state for a station

ht_capa HT capabilities of station

vht_capa VHT capabilities of station

uapsd_queues bitmap of queues configured for uapsd. same format as the AC bitmap in the QoS info field

max_sp max Service Period. same format as the MAX_SP in the QoS info field (but already shifted down)

local_pm local link-specific mesh power save mode (no change when set to unknown)

capability station capability

ext_capab extended capabilities of the station

ext_capab_len number of extended capabilities

supported_channels supported channels in IEEE 802.11 format

supported_channels_len number of supported channels

supported_oper_classes supported oper classes in IEEE 802.11 format

supported_oper_classes_len number of supported operating classes

opmode_notif operating mode field from Operating Mode Notification

opmode_notif_used information if operating mode field is used

support_p2p_ps information if station supports P2P PS mechanism

he_capa HE capabilities of station

he_capa_len the length of the HE capabilities

airtime_weight airtime scheduler weight for this station

txpwr transmit power for an associated station

he_6ghz_capa HE 6 GHz Band capabilities of station

Description

Used to change and create a new station.

enum **rate_info_flags**
bitrate info flags

Constants

RATE_INFO_FLAGS_MCS mcs field filled with HT MCS

RATE_INFO_FLAGS_VHT_MCS mcs field filled with VHT MCS

RATE_INFO_FLAGS_SHORT_GI 400ns guard interval

RATE_INFO_FLAGS_DMG 60GHz MCS

RATE_INFO_FLAGS_HE_MCS HE MCS information

RATE_INFO_FLAGS_EDMG 60GHz MCS in EDMG mode

Description

Used by the driver to indicate the specific rate transmission type for 802.11n transmissions.

struct **rate_info**
bitrate information

Definition

```
struct rate_info {
  u8 flags;
  u8 mcs;
  u16 legacy;
  u8 nss;
  u8 bw;
  u8 he_gi;
  u8 he_dcm;
  u8 he_ru_alloc;
  u8 n_bonded_ch;
};
```

Members

flags bitflag of flags from `enum rate_info_flags`

mcs mcs index if struct describes an HT/VHT/HE rate

legacy bitrate in 100kbit/s for 802.11abg

nss number of streams (VHT & HE only)

bw bandwidth (from `enum rate_info_bw`)

he_gi HE guard interval (from `enum nl80211_he_gi`)

he_dcm HE DCM value

he_ru_alloc HE RU allocation (from `enum nl80211_he_ru_alloc`, only valid if bw is `RATE_INFO_BW_HE_RU`)

n_bonded_ch In case of EDMG the number of bonded channels (1-4)

Description

Information about a receiving or transmitting bitrate

struct **station_info**
station information

Definition

```
struct station_info {
  u64 filled;
  u32 connected_time;
  u32 inactive_time;
  u64 assoc_at;
  u64 rx_bytes;
  u64 tx_bytes;
  u16 llid;
  u16 plid;
  u8 plink_state;
  s8 signal;
  s8 signal_avg;
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
  s8 chain_signal_avg[IEEE80211_MAX_CHAINS];
  struct rate_info txrate;
  struct rate_info rxrate;
  u32 rx_packets;
  u32 tx_packets;
  u32 tx_retries;
  u32 tx_failed;
  u32 rx_dropped_misc;
  struct sta_bss_parameters bss_param;
  struct nl80211_sta_flag_update sta_flags;
  int generation;
  const u8 *assoc_req_ies;
  size_t assoc_req_ies_len;
  u32 beacon_loss_count;
  s64 t_offset;
  enum nl80211_mesh_power_mode local_pm;
  enum nl80211_mesh_power_mode peer_pm;
  enum nl80211_mesh_power_mode nonpeer_pm;
  u32 expected_throughput;
  u64 tx_duration;
  u64 rx_duration;
  u64 rx_beacon;
  u8 rx_beacon_signal_avg;
  u8 connected_to_gate;
  struct cfg80211_tid_stats *pertid;
  s8 ack_signal;
  s8 avg_ack_signal;
  u16 airtime_weight;
  u32 rx_mpdu_count;
  u32 fcs_err_count;
```

(continues on next page)

(continued from previous page)

```
  u32 airtime_link_metric;
};
```

Members

filled bitflag of flags using the bits of `enum nl80211_sta_info` to indicate the relevant values in this struct for them

connected_time time(in secs) since a station is last connected

inactive_time time since last station activity (tx/rx) in milliseconds

assoc_at bootime (ns) of the last association

rx_bytes bytes (size of MPDUs) received from this station

tx_bytes bytes (size of MPDUs) transmitted to this station

llid mesh local link id

plid mesh peer link id

plink_state mesh peer link state

signal The signal strength, type depends on the wiphy's signal_type. For CFG80211_SIGNAL_TYPE_MBM, value is expressed in _dBm_.

signal_avg Average signal strength, type depends on the wiphy's signal_type. For CFG80211_SIGNAL_TYPE_MBM, value is expressed in _dBm_.

chains bitmask for filled values in **chain_signal**, **chain_signal_avg**

chain_signal per-chain signal strength of last received packet in dBm

chain_signal_avg per-chain signal strength average in dBm

txrate current unicast bitrate from this station

rxrate current unicast bitrate to this station

rx_packets packets (MSDUs & MMPDUs) received from this station

tx_packets packets (MSDUs & MMPDUs) transmitted to this station

tx_retries cumulative retry counts (MPDUs)

tx_failed number of failed transmissions (MPDUs) (retries exceeded, no ACK)

rx_dropped_misc Dropped for un-specified reason.

bss_param current BSS parameters

sta_flags station flags mask & values

generation generation number for nl80211 dumps. This number should increase every time the list of stations changes, i.e. when a station is added or removed, so that userspace can tell whether it got a consistent snapshot.

assoc_req_ies IEs from (Re)Association Request. This is used only when in AP mode with drivers that do not use user space MLME/SME implementation. The information is provided for the `cfg80211_new_sta()` calls to notify user space of the IEs.

assoc_req_ies_len Length of assoc_req_ies buffer in octets.

beacon_loss_count Number of times beacon loss event has triggered.

t_offset Time offset of the station relative to this host.

local_pm local mesh STA power save mode

peer_pm peer mesh STA power save mode

nonpeer_pm non-peer mesh STA power save mode

expected_throughput expected throughput in kbps (including 802.11 headers) towards this station.

tx_duration aggregate PPDU duration(usecs) for all the frames to a peer

rx_duration aggregate PPDU duration(usecs) for all the frames from a peer

rx_beacon number of beacons received from this peer

rx_beacon_signal_avg signal strength average (in dBm) for beacons received from this peer

connected_to_gate true if mesh STA has a path to mesh gate

pertid per-TID statistics, see `struct cfg80211_tid_stats`, using the last (IEEE80211_NUM_TIDS) index for MSDUs not encapsulated in QoS-MPDUs. Note that this doesn't use the **filled** bit, but is used if non-NULL.

ack_signal signal strength (in dBm) of the last ACK frame.

avg_ack_signal average rssi value of ack packet for the no of msdu's has been sent.

airtime_weight current airtime scheduling weight

rx_mpdu_count number of MPDUs received from this station

fcs_err_count number of packets (MPDUs) received from this station with an FCS error. This counter should be incremented only when TA of the received packet with an FCS error matches the peer MAC address.

airtime_link_metric mesh airtime link metric.

Description

Station information filled by driver for get_station() and dump_station.

enum **monitor_flags**
monitor flags

Constants

MONITOR_FLAG_CHANGED set if the flags were changed

MONITOR_FLAG_FCSFAIL pass frames with bad FCS

MONITOR_FLAG_PLCPFAIL pass frames with bad PLCP

MONITOR_FLAG_CONTROL pass control frames

MONITOR_FLAG_OTHER_BSS disable BSSID filtering

MONITOR_FLAG_COOK_FRAMES report frames after processing

MONITOR_FLAG_ACTIVE active monitor, ACKs frames on its MAC address

Description

Monitor interface configuration flags. Note that these must be the bits according to the nl80211 flags.

enum **mpath_info_flags**
mesh path information flags

Constants

MPATH_INFO_FRAME_QLEN **frame_qlen** filled

MPATH_INFO_SN **sn** filled

MPATH_INFO_METRIC **metric** filled

MPATH_INFO_EXPTIME **exptime** filled

MPATH_INFO_DISCOVERY_TIMEOUT **discovery_timeout** filled

MPATH_INFO_DISCOVERY_RETRIES **discovery_retries** filled

MPATH_INFO_FLAGS **flags** filled

MPATH_INFO_HOP_COUNT **hop_count** filled

MPATH_INFO_PATH_CHANGE **path_change_count** filled

Description

Used by the driver to indicate which info in `struct mpath_info` it has filled in during get_station() or dump_station().

struct **mpath_info**
mesh path information

Definition

```
struct mpath_info {
  u32 filled;
  u32 frame_qlen;
  u32 sn;
  u32 metric;
  u32 exptime;
  u32 discovery_timeout;
  u8 discovery_retries;
  u8 flags;
  u8 hop_count;
  u32 path_change_count;
  int generation;
};
```

Members

filled bitfield of flags from `enum mpath_info_flags`

frame_qlen number of queued frames for this destination

sn target sequence number

metric metric (cost) of this mesh path

exptime expiration time for the mesh path from now, in msecs

discovery_timeout total mesh path discovery timeout, in msecs

discovery_retries mesh path discovery retries

flags mesh path flags

hop_count hops to destination

path_change_count total number of path changes to destination

generation generation number for nl80211 dumps. This number should increase every time the list of mesh paths changes, i.e. when a station is added or removed, so that userspace can tell whether it got a consistent snapshot.

Description

Mesh path information filled by driver for get_mpath() and dump_mpath().

struct **bss_parameters**
BSS parameters

Definition

```
struct bss_parameters {
  int use_cts_prot;
  int use_short_preamble;
  int use_short_slot_time;
  const u8 *basic_rates;
  u8 basic_rates_len;
  int ap_isolate;
  int ht_opmode;
  s8 p2p_ctwindow, p2p_opp_ps;
};
```

Members

use_cts_prot Whether to use CTS protection (0 = no, 1 = yes, -1 = do not change)

use_short_preamble Whether the use of short preambles is allowed (0 = no, 1 = yes, -1 = do not change)

use_short_slot_time Whether the use of short slot time is allowed (0 = no, 1 = yes, -1 = do not change)

basic_rates basic rates in IEEE 802.11 format (or NULL for no change)

basic_rates_len number of basic rates

ap_isolate do not forward packets between connected stations

ht_opmode HT Operation mode (u16 = opmode, -1 = do not change)

p2p_ctwindow P2P CT Window (-1 = no change)

p2p_opp_ps P2P opportunistic PS (-1 = no change)

Description

Used to change BSS parameters (mainly for AP mode).

struct **ieee80211_txq_params**
TX queue parameters

Definition

```
struct ieee80211_txq_params {
  enum nl80211_ac ac;
  u16 txop;
  u16 cwmin;
  u16 cwmax;
  u8 aifs;
};
```

Members

ac AC identifier

txop Maximum burst time in units of 32 usecs, 0 meaning disabled

cwmin Minimum contention window [a value of the form 2^n-1 in the range 1..32767]

cwmax Maximum contention window [a value of the form 2^n-1 in the range 1..32767]

aifs Arbitration interframe space [0..255]

struct **cfg80211_crypto_settings**
Crypto settings

Definition

```
struct cfg80211_crypto_settings {
  u32 wpa_versions;
  u32 cipher_group;
  int n_ciphers_pairwise;
  u32 ciphers_pairwise[NL80211_MAX_NR_CIPHER_SUITES];
  int n_akm_suites;
  u32 akm_suites[NL80211_MAX_NR_AKM_SUITES];
  bool control_port;
  __be16 control_port_ethertype;
  bool control_port_no_encrypt;
  bool control_port_over_nl80211;
  bool control_port_no_preauth;
  struct key_params *wep_keys;
  int wep_tx_key;
  const u8 *psk;
  const u8 *sae_pwd;
  u8 sae_pwd_len;
};
```

Members

wpa_versions indicates which, if any, WPA versions are enabled (from enum nl80211_wpa_versions)

cipher_group group key cipher suite (or 0 if unset)

n_ciphers_pairwise number of AP supported unicast ciphers

ciphers_pairwise unicast key cipher suites

n_akm_suites number of AKM suites

akm_suites AKM suites

control_port Whether user space controls IEEE 802.1X port, i.e., sets/clears NL80211_STA_FLAG_AUTHORIZED. If true, the driver is required to assume that the port is unauthorized until authorized by user space. Otherwise, port is marked authorized by default.

control_port_ethertype the control port protocol that should be allowed through even on unauthorized ports

control_port_no_encrypt TRUE to prevent encryption of control port protocol frames.

control_port_over_nl80211 TRUE if userspace expects to exchange control port frames over NL80211 instead of the network interface.

control_port_no_preauth disables pre-auth rx over the nl80211 control port for mac80211

wep_keys static WEP keys, if not NULL points to an array of CFG80211_MAX_WEP_KEYS WEP keys

wep_tx_key key index (0..3) of the default TX static WEP key

psk PSK (for devices supporting 4-way-handshake offload)

sae_pwd password for SAE authentication (for devices supporting SAE offload)

sae_pwd_len length of SAE password (for devices supporting SAE offload)

struct **cfg80211_auth_request**
Authentication request data

Definition

```
struct cfg80211_auth_request {
  struct cfg80211_bss *bss;
  const u8 *ie;
  size_t ie_len;
  enum nl80211_auth_type auth_type;
  const u8 *key;
  u8 key_len, key_idx;
  const u8 *auth_data;
  size_t auth_data_len;
};
```

Members

bss The BSS to authenticate with, the callee must obtain a reference to it if it needs to keep it.

ie Extra IEs to add to Authentication frame or NULL

ie_len Length of ie buffer in octets

auth_type Authentication type (algorithm)

key WEP key for shared key authentication

key_len length of WEP key for shared key authentication

key_idx index of WEP key for shared key authentication

auth_data Fields and elements in Authentication frames. This contains the authentication frame body (non-IE and IE data), excluding the Authentication algorithm number, i.e., starting at the Authentication transaction sequence number field.

auth_data_len Length of auth_data buffer in octets

Description

This structure provides information needed to complete IEEE 802.11 authentication.

struct **cfg80211_assoc_request**
(Re)Association request data

Definition

```
struct cfg80211_assoc_request {
  struct cfg80211_bss *bss;
  const u8 *ie, *prev_bssid;
  size_t ie_len;
  struct cfg80211_crypto_settings crypto;
  bool use_mfp;
  u32 flags;
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
  struct ieee80211_vht_cap vht_capa, vht_capa_mask;
  const u8 *fils_kek;
  size_t fils_kek_len;
  const u8 *fils_nonces;
};
```

Members

bss The BSS to associate with. If the call is successful the driver is given a reference that it must give back to cfg80211_send_rx_assoc() or to `cfg80211_assoc_timeout()`. To ensure proper refcounting, new association requests while already associating must be rejected.

ie Extra IEs to add to (Re)Association Request frame or `NULL`

prev_bssid previous BSSID, if not `NULL` use reassociate frame. This is used to indicate a request to reassociate within the ESS instead of a request do the initial association with the ESS. When included, this is set to the BSSID of the current association, i.e., to the value that is included in the Current AP address field of the Reassociation Request frame.

ie_len Length of ie buffer in octets

crypto crypto settings

use_mfp Use management frame protection (IEEE 802.11w) in this association

flags See `enum cfg80211_assoc_req_flags`

ht_capa HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Un-supported values will be ignored.

ht_capa_mask The bits of ht_capa which are to be used.

vht_capa VHT capability override

vht_capa_mask VHT capability mask indicating which fields to use

fils_kek FILS KEK for protecting (Re)Association Request/Response frame or NULL if FILS is not used.

fils_kek_len Length of fils_kek in octets

fils_nonces FILS nonces (part of AAD) for protecting (Re)Association Request/Response frame or NULL if FILS is not used. This field starts with 16 octets of STA Nonce followed by 16 octets of AP Nonce.

Description

This structure provides information needed to complete IEEE 802.11 (re)association.

struct **cfg80211_deauth_request**
Deauthentication request data

Definition

```
struct cfg80211_deauth_request {
  const u8 *bssid;
  const u8 *ie;
  size_t ie_len;
  u16 reason_code;
  bool local_state_change;
};
```

Members

bssid the BSSID of the BSS to deauthenticate from

ie Extra IEs to add to Deauthentication frame or NULL

ie_len Length of ie buffer in octets

reason_code The reason code for the deauthentication

local_state_change if set, change local state only and do not set a deauth frame

Description

This structure provides information needed to complete IEEE 802.11 deauthentication.

struct **cfg80211_disassoc_request**
Disassociation request data

Definition

```
struct cfg80211_disassoc_request {
  struct cfg80211_bss *bss;
  const u8 *ie;
  size_t ie_len;
  u16 reason_code;
  bool local_state_change;
};
```

Members

bss the BSS to disassociate from

ie Extra IEs to add to Disassociation frame or NULL

ie_len Length of ie buffer in octets

reason_code The reason code for the disassociation

local_state_change This is a request for a local state only, i.e., no Disassociation frame is to be transmitted.

Description

This structure provides information needed to complete IEEE 802.11 disassociation.

struct **cfg80211_ibss_params**
IBSS parameters

Definition

```
struct cfg80211_ibss_params {
  const u8 *ssid;
  const u8 *bssid;
  struct cfg80211_chan_def chandef;
  const u8 *ie;
  u8 ssid_len, ie_len;
  u16 beacon_interval;
  u32 basic_rates;
  bool channel_fixed;
  bool privacy;
  bool control_port;
  bool control_port_over_nl80211;
  bool userspace_handles_dfs;
  int mcast_rate[NUM_NL80211_BANDS];
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
  struct key_params *wep_keys;
  int wep_tx_key;
};
```

Members

ssid The SSID, will always be non-null.

bssid Fixed BSSID requested, maybe be NULL, if set do not search for IBSSs with a different BSSID.

chandef defines the channel to use if no other IBSS to join can be found

ie information element(s) to include in the beacon

ssid_len The length of the SSID, will always be non-zero.

ie_len length of that

beacon_interval beacon interval to use

basic_rates bitmap of basic rates to use when creating the IBSS

channel_fixed The channel should be fixed – do not search for IBSSs to join on other channels.

privacy this is a protected network, keys will be configured after joining

control_port whether user space controls IEEE 802.1X port, i.e., sets/clears NL80211_STA_FLAG_AUTHORIZED. If true, the driver is required to assume that the port is unauthorized until authorized by user space. Otherwise, port is marked authorized by default.

control_port_over_nl80211 TRUE if userspace expects to exchange control port frames over NL80211 instead of the network interface.

userspace_handles_dfs whether user space controls DFS operation, i.e. changes the channel when a radar is detected. This is required to operate on DFS channels.

mcast_rate per-band multicast rate index + 1 (0: disabled)

ht_capa HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Un-supported values will be ignored.

ht_capa_mask The bits of ht_capa which are to be used.

wep_keys static WEP keys, if not NULL points to an array of CFG80211_MAX_WEP_KEYS WEP keys

wep_tx_key key index (0..3) of the default TX static WEP key

Description

This structure defines the IBSS parameters for the join_ibss() method.

struct **cfg80211_connect_params**
Connection parameters

Definition

```
struct cfg80211_connect_params {
  struct ieee80211_channel *channel;
  struct ieee80211_channel *channel_hint;
  const u8 *bssid;
  const u8 *bssid_hint;
  const u8 *ssid;
  size_t ssid_len;
  enum nl80211_auth_type auth_type;
  const u8 *ie;
  size_t ie_len;
  bool privacy;
  enum nl80211_mfp mfp;
  struct cfg80211_crypto_settings crypto;
  const u8 *key;
  u8 key_len, key_idx;
  u32 flags;
  int bg_scan_period;
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
  struct ieee80211_vht_cap vht_capa;
  struct ieee80211_vht_cap vht_capa_mask;
  bool pbss;
  struct cfg80211_bss_selection bss_select;
  const u8 *prev_bssid;
  const u8 *fils_erp_username;
  size_t fils_erp_username_len;
```

(continues on next page)

(continued from previous page)

```
  const u8 *fils_erp_realm;
  size_t fils_erp_realm_len;
  u16 fils_erp_next_seq_num;
  const u8 *fils_erp_rrk;
  size_t fils_erp_rrk_len;
  bool want_1x;
  struct ieee80211_edmg edmg;
};
```

Members

channel The channel to use or NULL if not specified (auto-select based on scan results)

channel_hint The channel of the recommended BSS for initial connection or NULL if not specified

bssid The AP BSSID or NULL if not specified (auto-select based on scan results)

bssid_hint The recommended AP BSSID for initial connection to the BSS or NULL if not specified. Unlike the **bssid** parameter, the driver is allowed to ignore this **bssid_hint** if it has knowledge of a better BSS to use.

ssid SSID

ssid_len Length of ssid in octets

auth_type Authentication type (algorithm)

ie IEs for association request

ie_len Length of assoc_ie in octets

privacy indicates whether privacy-enabled APs should be used

mfp indicate whether management frame protection is used

crypto crypto settings

key WEP key for shared key authentication

key_len length of WEP key for shared key authentication

key_idx index of WEP key for shared key authentication

flags See enum cfg80211_assoc_req_flags

bg_scan_period Background scan period in seconds or -1 to indicate that default value is to be used.

ht_capa HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Un-supported values will be ignored.

ht_capa_mask The bits of ht_capa which are to be used.

vht_capa VHT Capability overrides

vht_capa_mask The bits of vht_capa which are to be used.

pbss if set, connect to a PCP instead of AP. Valid for DMG networks.

bss_select criteria to be used for BSS selection.

prev_bssid previous BSSID, if not NULL use reassociate frame. This is used to indicate a request to reassociate within the ESS instead of a request do the initial association with the ESS. When included, this is set to the BSSID of the current association, i.e., to the value that is included in the Current AP address field of the Reassociation Request frame.

fils_erp_username EAP re-authentication protocol (ERP) username part of the NAI or NULL if not specified. This is used to construct FILS wrapped data IE.

fils_erp_username_len Length of **fils_erp_username** in octets.

fils_erp_realm EAP re-authentication protocol (ERP) realm part of NAI or NULL if not specified. This specifies the domain name of ER server and is used to construct FILS wrapped data IE.

fils_erp_realm_len Length of **fils_erp_realm** in octets.

fils_erp_next_seq_num The next sequence number to use in the FILS ERP messages. This is also used to construct FILS wrapped data IE.

fils_erp_rrk ERP re-authentication Root Key (rRK) used to derive additional keys in FILS or NULL if not specified.

fils_erp_rrk_len Length of **fils_erp_rrk** in octets.

want_1x indicates user-space supports and wants to use 802.1X driver offload of 4-way handshake.

edmg define the EDMG channels. This may specify multiple channels and bonding options for the driver to choose from, based on BSS configuration.

Description

This structure provides information needed to complete IEEE 802.11 authentication and association.

struct **cfg80211_pmksa**
PMK Security Association

Definition

```
struct cfg80211_pmksa {
  const u8 *bssid;
  const u8 *pmkid;
  const u8 *pmk;
  size_t pmk_len;
  const u8 *ssid;
  size_t ssid_len;
  const u8 *cache_id;
  u32 pmk_lifetime;
  u8 pmk_reauth_threshold;
};
```

Members

bssid The AP's BSSID (may be NULL).

pmkid The identifier to refer a PMKSA.

pmk The PMK for the PMKSA identified by **pmkid**. This is used for key derivation by a FILS STA. Otherwise, NULL.

pmk_len Length of the **pmk**. The length of **pmk** can differ depending on the hash algorithm used to generate this.

ssid SSID to specify the ESS within which a PMKSA is valid when using FILS cache identifier (may be NULL).

ssid_len Length of the **ssid** in octets.

cache_id 2-octet cache identifier advertized by a FILS AP identifying the scope of PMKSA. This is valid only if **ssid_len** is non-zero (may be NULL).

pmk_lifetime Maximum lifetime for PMKSA in seconds (dot11RSNAConfigPMKLifetime) or 0 if not specified. The configured PMKSA must not be used for PMKSA caching after expiration and any keys derived from this PMK become invalid on expiration, i.e., the current association must be dropped if the PMK used for it expires.

pmk_reauth_threshold Threshold time for reauthentication (percentage of PMK lifetime, dot11RSNAConfigPMKReauthThreshold) or 0 if not specified. Drivers are expected to trigger a full authentication instead of using this PMKSA for caching when reassociating to a new BSS after this threshold to generate a new PMK before the current one expires.

Description

This structure is passed to the set/del_pmksa() method for PMKSA caching.

void **cfg80211_rx_mlme_mgmt**(struct net_device * dev, const u8 * buf, size_t len)
notification of processed MLME management frame

Parameters

struct net_device * dev network device

const u8 * buf authentication frame (header + body)

size_t len length of the frame data

Description

This function is called whenever an authentication, disassociation or deauthentication frame has been received and processed in station mode. After being asked to authenticate via cfg80211_ops::auth() the driver must call either this function or cfg80211_auth_timeout(). After being asked to associate via cfg80211_ops::assoc() the driver must call either this function or cfg80211_auth_timeout(). While connected, the driver must calls this for received and processed disassociation and deauthentication frames. If the frame couldn't be used because it was unprotected, the driver must call the function cfg80211_rx_unprot_mlme_mgmt() instead.

This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_auth_timeout**(struct net_device * dev, const u8 * addr)
notification of timed out authentication

Parameters

struct net_device * dev network device

const u8 * addr The MAC address of the device with which the authentication timed out

Description

This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_rx_assoc_resp**(struct net_device * dev, struct cfg80211_bss * bss, const u8 * buf, size_t len, int uapsd_queues, const u8 * req_ies, size_t req_ies_len)
notification of processed association response

Parameters

struct net_device * dev network device

struct cfg80211_bss * bss the BSS that association was requested with, ownership of the pointer moves to cfg80211 in this call

const u8 * buf (Re)Association Response frame (header + body)

size_t len length of the frame data

int uapsd_queues bitmap of queues configured for uapsd. Same format as the AC bitmap in the QoS info field

const u8 * req_ies information elements from the (Re)Association Request frame

size_t req_ies_len length of req_ies data

Description

After being asked to associate via cfg80211_ops::assoc() the driver must call either this function or `cfg80211_auth_timeout()`.

This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_assoc_timeout**(struct net_device * dev, struct cfg80211_bss * bss)
notification of timed out association

Parameters

struct net_device * dev network device

struct cfg80211_bss * bss The BSS entry with which association timed out.

Description

This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_tx_mlme_mgmt**(struct net_device * dev, const u8 * buf, size_t len)
notification of transmitted deauth/disassoc frame

Parameters

struct net_device * dev network device

const u8 * buf 802.11 frame (header + body)

size_t len length of the frame data

Description

This function is called whenever deauthentication has been processed in station mode. This includes both received deauthentication frames and locally generated ones. This function may sleep. The caller must hold the corresponding wdev's mutex.

void **cfg80211_ibss_joined**(struct net_device * dev, const u8 * bssid, struct ieee80211_channel * channel, gfp_t gfp)
notify cfg80211 that device joined an IBSS

Parameters

struct net_device * dev network device

const u8 * bssid the BSSID of the IBSS joined

struct ieee80211_channel * channel the channel of the IBSS joined

gfp_t gfp allocation flags

Description

This function notifies cfg80211 that the device joined an IBSS or switched to a different BSSID. Before this function can be called, either a beacon has to have been received from the IBSS, or one of the cfg80211_inform_bss{,_frame} functions must have been called with the locally generated beacon – this guarantees that there is always a scan result for this IBSS. cfg80211 will handle the rest.

struct **cfg80211_connect_resp_params**
Connection response params

Definition

```
struct cfg80211_connect_resp_params {
  int status;
  const u8 *bssid;
  struct cfg80211_bss *bss;
  const u8 *req_ie;
  size_t req_ie_len;
  const u8 *resp_ie;
  size_t resp_ie_len;
  struct cfg80211_fils_resp_params fils;
  enum nl80211_timeout_reason timeout_reason;
};
```

Members

status Status code, `WLAN_STATUS_SUCCESS` for successful connection, use `WLAN_STATUS_UNSPECIFIED_FAILURE` if your device cannot give you the real status code for failures. If this call is used to report a failure due to a timeout (e.g., not receiving an Authentication frame from the AP) instead of an explicit rejection by the AP, -1 is used to indicate that this is a failure, but without a status code. **timeout_reason** is used to report the reason for the timeout in that case.

bssid The BSSID of the AP (may be `NULL`)

bss Entry of bss to which STA got connected to, can be obtained through cfg80211_get_bss() (may be `NULL`). But it is recommended to store the bss

from the connect_request and hold a reference to it and return through this param to avoid a warning if the bss is expired during the connection, esp. for those drivers implementing connect op. Only one parameter among **bssid** and **bss** needs to be specified.

req_ie Association request IEs (may be NULL)

req_ie_len Association request IEs length

resp_ie Association response IEs (may be NULL)

resp_ie_len Association response IEs length

fils FILS connection response parameters.

timeout_reason Reason for connection timeout. This is used when the connection fails due to a timeout instead of an explicit rejection from the AP. NL80211_TIMEOUT_UNSPECIFIED is used when the timeout reason is not known. This value is used only if **status** < 0 to indicate that the failure is due to a timeout and not due to explicit rejection by the AP. This value is ignored in other cases (**status** >= 0).

void **cfg80211_connect_done**(struct net_device * dev, struct cfg80211_connect_resp_params * params, gfp_t gfp)

notify cfg80211 of connection result

Parameters

struct net_device * dev network device

struct cfg80211_connect_resp_params * params connection response parameters

gfp_t gfp allocation flags

Description

It should be called by the underlying driver once execution of the connection request from connect() has been completed. This is similar to cfg80211_connect_bss(), but takes a structure pointer for connection response parameters. Only one of the functions among cfg80211_connect_bss(), cfg80211_connect_result(), cfg80211_connect_timeout(), and cfg80211_connect_done() should be called.

void **cfg80211_connect_result**(struct net_device * dev, const u8 * bssid, const u8 * req_ie, size_t req_ie_len, const u8 * resp_ie, size_t resp_ie_len, u16 status, gfp_t gfp)

notify cfg80211 of connection result

Parameters

struct net_device * dev network device

const u8 * bssid the BSSID of the AP

const u8 * req_ie association request IEs (maybe be NULL)

size_t req_ie_len association request IEs length

const u8 * resp_ie association response IEs (may be NULL)

size_t resp_ie_len assoc response IEs length

u16 status status code, WLAN_STATUS_SUCCESS for successful connection, use WLAN_STATUS_UNSPECIFIED_FAILURE if your device cannot give you the real status code for failures.

gfp_t gfp allocation flags

Description

It should be called by the underlying driver once execution of the connection request from connect() has been completed. This is similar to cfg80211_connect_bss() which allows the exact bss entry to be specified. Only one of the functions among cfg80211_connect_bss(), cfg80211_connect_result(), cfg80211_connect_timeout(), and cfg80211_connect_done() should be called.

void **cfg80211_connect_bss**(struct net_device * dev, const u8 * bssid, struct cfg80211_bss * bss, const u8 * req_ie, size_t req_ie_len, const u8 * resp_ie, size_t resp_ie_len, int status, gfp_t gfp, enum nl80211_timeout_reason timeout_reason)

notify cfg80211 of connection result

Parameters

struct net_device * dev network device

const u8 * bssid the BSSID of the AP

struct cfg80211_bss * bss Entry of bss to which STA got connected to, can be obtained through cfg80211_get_bss() (may be NULL). But it is recommended to store the bss from the connect_request and hold a reference to it and return through this param to avoid a warning if the bss is expired during the connection, esp. for those drivers implementing connect op. Only one parameter among **bssid** and **bss** needs to be specified.

const u8 * req_ie association request IEs (maybe be NULL)

size_t req_ie_len association request IEs length

const u8 * resp_ie association response IEs (may be NULL)

size_t resp_ie_len assoc response IEs length

int status status code, WLAN_STATUS_SUCCESS for successful connection, use WLAN_STATUS_UNSPECIFIED_FAILURE if your device cannot give you the real status code for failures. If this call is used to report a failure due to a timeout (e.g., not receiving an Authentication frame from the AP) instead of an explicit rejection by the AP, -1 is used to indicate that this is a failure, but without a status code. **timeout_reason** is used to report the reason for the timeout in that case.

gfp_t gfp allocation flags

enum nl80211_timeout_reason timeout_reason reason for connection timeout. This is used when the connection fails due to a timeout instead of an explicit

rejection from the AP. NL80211_TIMEOUT_UNSPECIFIED is used when the timeout reason is not known. This value is used only if **status** < 0 to indicate that the failure is due to a timeout and not due to explicit rejection by the AP. This value is ignored in other cases (**status** >= 0).

Description

It should be called by the underlying driver once execution of the connection request from connect() has been completed. This is similar to cfg80211_connect_result(), but with the option of identifying the exact bss entry for the connection. Only one of the functions among cfg80211_connect_bss(), cfg80211_connect_result(), cfg80211_connect_timeout(), and cfg80211_connect_done() should be called.

void **cfg80211_connect_timeout**(struct net_device * dev, const u8 * bssid, const u8 * req_ie, size_t req_ie_len, gfp_t gfp, enum nl80211_timeout_reason timeout_reason)

notify cfg80211 of connection timeout

Parameters

struct net_device * dev network device

const u8 * bssid the BSSID of the AP

const u8 * req_ie association request IEs (maybe be NULL)

size_t req_ie_len association request IEs length

gfp_t gfp allocation flags

enum nl80211_timeout_reason timeout_reason reason for connection timeout.

Description

It should be called by the underlying driver whenever connect() has failed in a sequence where no explicit authentication/association rejection was received from the AP. This could happen, e.g., due to not being able to send out the Authentication or Association Request frame or timing out while waiting for the response. Only one of the functions among cfg80211_connect_bss(), cfg80211_connect_result(), cfg80211_connect_timeout(), and cfg80211_connect_done() should be called.

void **cfg80211_roamed**(struct net_device * dev, struct cfg80211_roam_info * info, gfp_t gfp)

notify cfg80211 of roaming

Parameters

struct net_device * dev network device

struct cfg80211_roam_info * info information about the new BSS. struct cfg80211_roam_info.

gfp_t gfp allocation flags

Description

This function may be called with the driver passing either the BSSID of the new AP or passing the bss entry to avoid a race in timeout of the bss entry. It should be called by the underlying driver whenever it roamed from one AP to another while connected. Drivers which have roaming implemented in firmware should pass the bss entry to avoid a race in bss entry timeout where the bss entry of the new AP is seen in the driver, but gets timed out by the time it is accessed in __cfg80211_roamed() due to delay in scheduling rdev->event_work. In case of any failures, the reference is released either in `cfg80211_roamed()` or in __cfg80211_romed(), Otherwise, it will be released while disconnecting from the current bss.

void **cfg80211_disconnected**(struct net_device * dev, u16 reason, const u8 * ie, size_t ie_len, bool locally_generated, gfp_t gfp)

notify cfg80211 that connection was dropped

Parameters

struct net_device * dev network device

u16 reason reason code for the disconnection, set it to 0 if unknown

const u8 * ie information elements of the deauth/disassoc frame (may be `NULL`)

size_t ie_len length of IEs

bool locally_generated disconnection was requested locally

gfp_t gfp allocation flags

Description

After it calls this function, the driver should enter an idle state and not try to connect to any AP any more.

void **cfg80211_ready_on_channel**(struct wireless_dev * wdev, u64 cookie, struct ieee80211_channel * chan, unsigned int duration, gfp_t gfp)

notification of remain_on_channel start

Parameters

struct wireless_dev * wdev wireless device

u64 cookie the request cookie

struct ieee80211_channel * chan The current channel (from remain_on_channel request)

unsigned int duration Duration in milliseconds that the driver intents to remain on the channel

gfp_t gfp allocation flags

void **cfg80211_remain_on_channel_expired**(struct wireless_dev * wdev, u64 cookie, struct ieee80211_channel * chan, gfp_t gfp)

remain_on_channel duration expired

Parameters

struct wireless_dev * wdev wireless device

u64 cookie the request cookie

struct ieee80211_channel * chan The current channel (from remain_on_channel request)

gfp_t gfp allocation flags

void **cfg80211_new_sta**(struct net_device * dev, const u8 * mac_addr, struct station_info * sinfo, gfp_t gfp)
notify userspace about station

Parameters

struct net_device * dev the netdev

const u8 * mac_addr the station's address

struct station_info * sinfo the station information

gfp_t gfp allocation flags

bool **cfg80211_rx_mgmt**(struct wireless_dev * wdev, int freq, int sig_dbm, const u8 * buf, size_t len, u32 flags)
notification of received, unprocessed management frame

Parameters

struct wireless_dev * wdev wireless device receiving the frame

int freq Frequency on which the frame was received in MHz

int sig_dbm signal strength in dBm, or 0 if unknown

const u8 * buf Management frame (header + body)

size_t len length of the frame data

u32 flags flags, as defined in enum nl80211_rxmgmt_flags

Description

This function is called whenever an Action frame is received for a station mode interface, but is not processed in kernel.

Return

`true` if a user space application has registered for this frame. For action frames, that makes it responsible for rejecting unrecognized action frames; `false` otherwise, in which case for action frames the driver is responsible for rejecting the frame.

void **cfg80211_mgmt_tx_status**(struct wireless_dev * wdev, u64 cookie, const u8 * buf, size_t len, bool ack, gfp_t gfp)
notification of TX status for management frame

Parameters

struct wireless_dev * wdev wireless device receiving the frame

u64 cookie Cookie returned by cfg80211_ops::mgmt_tx()

const u8 * buf Management frame (header + body)

size_t len length of the frame data

bool ack Whether frame was acknowledged

gfp_t gfp context flags

Description

This function is called whenever a management frame was requested to be transmitted with cfg80211_ops::mgmt_tx() to report the TX status of the transmission attempt.

void **cfg80211_cqm_rssi_notify**(struct net_device * dev, enum nl80211_cqm_rssi_threshold_event rssi_event, s32 rssi_level, gfp_t gfp)
connection quality monitoring rssi event

Parameters

struct net_device * dev network device

enum nl80211_cqm_rssi_threshold_event rssi_event the triggered RSSI event

s32 rssi_level new RSSI level value or 0 if not available

gfp_t gfp context flags

Description

This function is called when a configured connection quality monitoring rssi threshold reached event occurs.

void **cfg80211_cqm_pktloss_notify**(struct net_device * dev, const u8 * peer, u32 num_packets, gfp_t gfp)
notify userspace about packetloss to peer

Parameters

struct net_device * dev network device

const u8 * peer peer's MAC address

u32 num_packets how many packets were lost – should be a fixed threshold but probably no less than maybe 50, or maybe a throughput dependent threshold (to account for temporary interference)

gfp_t gfp context flags

void **cfg80211_michael_mic_failure**(struct net_device * dev, const u8 * addr, enum nl80211_key_type key_type, int key_id, const u8 * tsc, gfp_t gfp)
notification of Michael MIC failure (TKIP)

Parameters

struct net_device * dev network device

const u8 * addr The source MAC address of the frame

enum nl80211_key_type key_type The key type that the received frame used

int key_id Key identifier (0..3). Can be -1 if missing.

const u8 * tsc The TSC value of the frame that generated the MIC failure (6 octets)

gfp_t gfp allocation flags

Description

This function is called whenever the local MAC detects a MIC failure in a received frame. This matches with MLME-MICHAELMICFAILURE.indication() primitive.

47.2.3 Scanning and BSS list handling

The scanning process itself is fairly simple, but cfg80211 offers quite a bit of helper functionality. To start a scan, the scan operation will be invoked with a scan definition. This scan definition contains the channels to scan, and the SSIDs to send probe requests for (including the wildcard, if desired). A passive scan is indicated by having no SSIDs to probe. Additionally, a scan request may contain extra information elements that should be added to the probe request. The IEs are guaranteed to be well-formed, and will not exceed the maximum length the driver advertised in the wiphy structure.

When scanning finds a BSS, cfg80211 needs to be notified of that, because it is responsible for maintaining the BSS list; the driver should not maintain a list itself. For this notification, various functions exist.

Since drivers do not maintain a BSS list, there are also a number of functions to search for a BSS and obtain information about it from the BSS structure cfg80211 maintains. The BSS list is also made available to userspace.

struct **cfg80211_ssid**
SSID description

Definition

```
struct cfg80211_ssid {
  u8 ssid[IEEE80211_MAX_SSID_LEN];
  u8 ssid_len;
};
```

Members

ssid the SSID

ssid_len length of the ssid

struct **cfg80211_scan_request**
scan request description

Definition

```
struct cfg80211_scan_request {
  struct cfg80211_ssid *ssids;
  int n_ssids;
  u32 n_channels;
  enum nl80211_bss_scan_width scan_width;
  const u8 *ie;
  size_t ie_len;
```

(continues on next page)

(continued from previous page)

```
  u16 duration;
  bool duration_mandatory;
  u32 flags;
  u32 rates[NUM_NL80211_BANDS];
  struct wireless_dev *wdev;
  u8 mac_addr[ETH_ALEN] ;
  u8 mac_addr_mask[ETH_ALEN] ;
  u8 bssid[ETH_ALEN] ;
  struct wiphy *wiphy;
  unsigned long scan_start;
  struct cfg80211_scan_info info;
  bool notified;
  bool no_cck;
  struct ieee80211_channel *channels[];
};
```

Members

ssids SSIDs to scan for (active scan only)

n_ssids number of SSIDs

n_channels total number of channels to scan

scan_width channel width for scanning

ie optional information element(s) to add into Probe Request or NULL

ie_len length of ie in octets

duration how long to listen on each channel, in TUs. If duration_mandatory is not set, this is the maximum dwell time and the actual dwell time may be shorter.

duration_mandatory if set, the scan duration must be as specified by the duration field.

flags bit field of flags controlling operation

rates bitmap of rates to advertise for each band

wdev the wireless device to scan for

mac_addr MAC address used with randomisation

mac_addr_mask MAC address mask used with randomisation, bits that are 0 in the mask should be randomised, bits that are 1 should be taken from the **mac_addr**

bssid BSSID to scan for (most commonly, the wildcard BSSID)

wiphy the wiphy this was for

scan_start time (in jiffies) when the scan started

info (internal) information about completed scan

notified (internal) scan request was notified as done or aborted

no_cck used to send probe requests at non CCK rate in 2GHz band

channels channels to scan on.

void **cfg80211_scan_done**(struct cfg80211_scan_request * request, struct cfg80211_scan_info * info)
notify that scan finished

Parameters

struct cfg80211_scan_request * request the corresponding scan request

struct cfg80211_scan_info * info information about the completed scan

struct **cfg80211_bss**
BSS description

Definition

```
struct cfg80211_bss {
  struct ieee80211_channel *channel;
  enum nl80211_bss_scan_width scan_width;
  const struct cfg80211_bss_ies __rcu *ies;
  const struct cfg80211_bss_ies __rcu *beacon_ies;
  const struct cfg80211_bss_ies __rcu *proberesp_ies;
  struct cfg80211_bss *hidden_beacon_bss;
  struct cfg80211_bss *transmitted_bss;
  struct list_head nontrans_list;
  s32 signal;
  u16 beacon_interval;
  u16 capability;
  u8 bssid[ETH_ALEN];
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
  u8 bssid_index;
  u8 max_bssid_indicator;
  u8 priv[] ;
};
```

Members

channel channel this BSS is on

scan_width width of the control channel

ies the information elements (Note that there is no guarantee that these are well-formed!); this is a pointer to either the beacon_ies or proberesp_ies depending on whether Probe Response frame has been received. It is always non-NULL.

beacon_ies the information elements from the last Beacon frame (implementation note: if **hidden_beacon_bss** is set this struct doesn't own the beacon_ies, but they're just pointers to the ones from the **hidden_beacon_bss** struct)

proberesp_ies the information elements from the last Probe Response frame

hidden_beacon_bss in case this BSS struct represents a probe response from a BSS that hides the SSID in its beacon, this points to the BSS struct that holds the beacon data. **beacon_ies** is still valid, of course, and points to the same data as hidden_beacon_bss->beacon_ies in that case.

transmitted_bss pointer to the transmitted BSS, if this is a non-transmitted one (multi-BSSID support)

nontrans_list list of non-transmitted BSS, if this is a transmitted one (multi-BSSID support)

signal signal strength value (type depends on the wiphy's signal_type)

beacon_interval the beacon interval as from the frame

capability the capability field in host byte order

bssid BSSID of the BSS

chains bitmask for filled values in **chain_signal**.

chain_signal per-chain signal strength of last received BSS in dBm.

bssid_index index in the multiple BSS set

max_bssid_indicator max number of members in the BSS set

priv private area for driver use, has at least wiphy->bss_priv_size bytes

Description

This structure describes a BSS (which may also be a mesh network) for use in scan results and similar.

struct **cfg80211_inform_bss**
BSS inform data

Definition

```
struct cfg80211_inform_bss {
  struct ieee80211_channel *chan;
  enum nl80211_bss_scan_width scan_width;
  s32 signal;
  u64 boottime_ns;
  u64 parent_tsf;
  u8 parent_bssid[ETH_ALEN] ;
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
};
```

Members

chan channel the frame was received on

scan_width scan width that was used

signal signal strength value, according to the wiphy's signal type

boottime_ns timestamp (CLOCK_BOOTTIME) when the information was received; should match the time when the frame was actually received by the device (not just by the host, in case it was buffered on the device) and be accurate to about 10ms. If the frame isn't buffered, just passing the return value of ktime_get_boottime_ns() is likely appropriate.

parent_tsf the time at the start of reception of the first octet of the timestamp field of the frame. The time is the TSF of the BSS specified by `parent_bssid`.

parent_bssid the BSS according to which `parent_tsf` is set. This is set to the BSS that requested the scan in which the beacon/probe was received.

chains bitmask for filled values in **chain_signal**.

chain_signal per-chain signal strength of last received BSS in dBm.

struct cfg80211_bss * **cfg80211_inform_bss_frame_data**(struct wiphy * wiphy, struct cfg80211_inform_bss * data, struct ieee80211_mgmt * mgmt, size_t len, gfp_t gfp)

inform cfg80211 of a received BSS frame

Parameters

struct wiphy * wiphy the wiphy reporting the BSS

struct cfg80211_inform_bss * data the BSS metadata

struct ieee80211_mgmt * mgmt the management frame (probe response or beacon)

size_t len length of the management frame

gfp_t gfp context flags

Description

This informs cfg80211 that BSS information was found and the BSS should be updated/added.

Return

A referenced struct, must be released with cfg80211_put_bss()! Or NULL on error.

struct cfg80211_bss * **cfg80211_inform_bss_data**(struct wiphy * wiphy, struct cfg80211_inform_bss * data, enum cfg80211_bss_frame_type ftype, const u8 * bssid, u64 tsf, u16 capability, u16 beacon_interval, const u8 * ie, size_t ielen, gfp_t gfp)

inform cfg80211 of a new BSS

Parameters

struct wiphy * wiphy the wiphy reporting the BSS

struct cfg80211_inform_bss * data the BSS metadata

enum cfg80211_bss_frame_type ftype frame type (if known)

const u8 * bssid the BSSID of the BSS

u64 tsf the TSF sent by the peer in the beacon/probe response (or 0)

u16 capability the capability field sent by the peer

u16 beacon_interval the beacon interval announced by the peer

const u8 * ie additional IEs sent by the peer

size_t ielen length of the additional IEs

gfp_t gfp context flags

Description

This informs cfg80211 that BSS information was found and the BSS should be updated/added.

Return

A referenced struct, must be released with cfg80211_put_bss()! Or NULL on error.

void **cfg80211_unlink_bss**(struct wiphy * wiphy, struct cfg80211_bss * bss)
unlink BSS from internal data structures

Parameters

struct wiphy * wiphy the wiphy

struct cfg80211_bss * bss the bss to remove

Description

This function removes the given BSS from the internal data structures thereby making it no longer show up in scan results etc. Use this function when you detect a BSS is gone. Normally BSSes will also time out, so it is not necessary to use this function at all.

const u8 * **cfg80211_find_ie**(u8 eid, const u8 * ies, int len)
find information element in data

Parameters

u8 eid element ID

const u8 * ies data consisting of IEs

int len length of data

Return

NULL if the element ID could not be found or if the element is invalid (claims to be longer than the given data), or a pointer to the first byte of the requested element, that is the byte containing the element ID.

Note

There are no checks on the element length other than having to fit into the given data.

const u8 * **ieee80211_bss_get_ie**(struct cfg80211_bss * bss, u8 id)
find IE with given ID

Parameters

struct cfg80211_bss * bss the bss to search

u8 id the element ID

Description

Note that the return value is an RCU-protected pointer, so rcu_read_lock() must be held when calling this function.

Return

`NULL` if not found.

47.2.4 Utility functions

cfg80211 offers a number of utility functions that can be useful.

int **ieee80211_channel_to_frequency**(int chan, enum nl80211_band band)
convert channel number to frequency

Parameters

int chan channel number

enum nl80211_band band band, necessary due to channel number overlap

Return

The corresponding frequency (in MHz), or 0 if the conversion failed.

int **ieee80211_frequency_to_channel**(int freq)
convert frequency to channel number

Parameters

int freq center frequency in MHz

Return

The corresponding channel, or 0 if the conversion failed.

struct ieee80211_channel * **ieee80211_get_channel**(struct wiphy * wiphy, int freq)
get channel struct from wiphy for specified frequency

Parameters

struct wiphy * wiphy the struct wiphy to get the channel for

int freq the center frequency (in MHz) of the channel

Return

The channel struct from **wiphy** at **freq**.

struct ieee80211_rate * **ieee80211_get_response_rate**(struct ieee80211_supported_band * sband, u32 basic_rates, int bitrate)
get basic rate for a given rate

Parameters

struct ieee80211_supported_band * sband the band to look for rates in

u32 basic_rates bitmap of basic rates

int bitrate the bitrate for which to find the basic rate

Return

The basic rate corresponding to a given bitrate, that is the next lower bitrate contained in the basic rate map, which is, for this function, given as a bitmap of indices of rates in the band's bitrate table.

unsigned int __attribute_const__ **ieee80211_hdrlen**(__le16 fc)
get header length in bytes from frame control

Parameters

__le16 fc frame control field in little-endian format

Return

The header length in bytes.

unsigned int **ieee80211_get_hdrlen_from_skb**(const struct sk_buff * skb)
get header length from data

Parameters

const struct sk_buff * skb the frame

Description

Given an skb with a raw 802.11 header at the data pointer this function returns the 802.11 header length.

Return

The 802.11 header length in bytes (not including encryption headers). Or 0 if the data in the sk_buff is too short to contain a valid 802.11 header.

struct **ieee80211_radiotap_iterator**
tracks walk thru present radiotap args

Definition

```
struct ieee80211_radiotap_iterator {
  struct ieee80211_radiotap_header *_rtheader;
  const struct ieee80211_radiotap_vendor_namespaces *_vns;
  const struct ieee80211_radiotap_namespace *current_namespace;
  unsigned char *_arg, *_next_ns_data;
  __le32 *_next_bitmap;
  unsigned char *this_arg;
  int this_arg_index;
  int this_arg_size;
  int is_radiotap_ns;
  int _max_length;
  int _arg_index;
  uint32_t _bitmap_shifter;
  int _reset_on_ext;
};
```

Members

_rtheader pointer to the radiotap header we are walking through

_vns vendor namespace definitions

current_namespace pointer to the current namespace definition (or internally NULL if the current namespace is unknown)

_arg next argument pointer

_next_ns_data beginning of the next namespace's data

_next_bitmap internal pointer to next present u32

this_arg pointer to current radiotap arg; it is valid after each call to ieee80211_radiotap_iterator_next() but also after ieee80211_radiotap_iterator_init() where it will point to the beginning of the actual data portion

this_arg_index index of current arg, valid after each successful call to ieee80211_radiotap_iterator_next()

this_arg_size length of the current arg, for convenience

is_radiotap_ns indicates whether the current namespace is the default radiotap namespace or not

_max_length length of radiotap header in cpu byte ordering

_arg_index next argument index

_bitmap_shifter internal shifter for curr u32 bitmap, b0 set == arg present

_reset_on_ext internal; reset the arg index to 0 when going to the next bitmap word

Description

Describes the radiotap parser state. Fields prefixed with an underscore must not be used by users of the parser, only by the parser internally.

47.2.5 Data path helpers

In addition to generic utilities, cfg80211 also offers functions that help implement the data path for devices that do not do the 802.11/802.3 conversion on the device.

int **ieee80211_data_to_8023**(struct sk_buff * skb, const u8 * addr, enum nl80211_iftype iftype)
convert an 802.11 data frame to 802.3

Parameters

struct sk_buff * skb the 802.11 data frame

const u8 * addr the device MAC address

enum nl80211_iftype iftype the virtual interface type

Return

0 on success. Non-zero on error.

void **ieee80211_amsdu_to_8023s**(struct sk_buff * skb, struct sk_buff_head * list, const u8 * addr, enum nl80211_iftype iftype, const unsigned int extra_headroom, const u8 * check_da, const u8 * check_sa)
decode an IEEE 802.11n A-MSDU frame

Parameters

struct sk_buff * skb The input A-MSDU frame without any headers.

struct sk_buff_head * list The output list of 802.3 frames. It must be allocated and initialized by by the caller.

const u8 * addr The device MAC address.

enum nl80211_iftype iftype The device interface type.

const unsigned int extra_headroom The hardware extra headroom for SKBs in the **list**.

const u8 * check_da DA to check in the inner ethernet header, or NULL

const u8 * check_sa SA to check in the inner ethernet header, or NULL

Description

Decode an IEEE 802.11 A-MSDU and convert it to a list of 802.3 frames. The **list** will be empty if the decode fails. The **skb** must be fully header-less before being passed in here; it is freed in this function.

unsigned int **cfg80211_classify8021d**(struct sk_buff * skb, struct cfg80211_qos_map * qos_map)
determine the 802.1p/1d tag for a data frame

Parameters

struct sk_buff * skb the data frame

struct cfg80211_qos_map * qos_map Interworking QoS mapping or `NULL` if not in use

Return

The 802.1p/1d tag.

47.2.6 Regulatory enforcement infrastructure

TODO

int **regulatory_hint**(struct wiphy * wiphy, const char * alpha2)
driver hint to the wireless core a regulatory domain

Parameters

struct wiphy * wiphy the wireless device giving the hint (used only for reporting conflicts)

const char * alpha2 the ISO/IEC 3166 alpha2 the driver claims its regulatory domain should be in. If **rd** is set this should be NULL. Note that if you set this to NULL you should still set rd->alpha2 to some accepted alpha2.

Description

Wireless drivers can use this function to hint to the wireless core what it believes should be the current regulatory domain by giving it an ISO/IEC 3166 alpha2 country code it knows its regulatory domain should be in or by providing a completely build regulatory domain. If the driver provides an ISO/IEC 3166 alpha2 userspace will be queried for a regulatory domain structure for the respective country.

The wiphy must have been registered to cfg80211 prior to this call. For cfg80211 drivers this means you must first use `wiphy_register()`, for mac80211 drivers you must first use `ieee80211_register_hw()`.

Drivers should check the return value, its possible you can get an -ENOMEM.

Return

0 on success. -ENOMEM.

void **wiphy_apply_custom_regulatory**(struct wiphy * wiphy, const struct ieee80211_regdomain * regd)
apply a custom driver regulatory domain

Parameters

struct wiphy * wiphy the wireless device we want to process the regulatory domain on

const struct ieee80211_regdomain * regd the custom regulatory domain to use for this wiphy

Description

Drivers can sometimes have custom regulatory domains which do not apply to a specific country. Drivers can use this to apply such custom regulatory domains. This routine must be called prior to wiphy registration. The custom regulatory domain will be trusted completely and as such previous default channel settings will be disregarded. If no rule is found for a channel on the regulatory domain the channel will be disabled. Drivers using this for a wiphy should also set the wiphy flag REGULATORY_CUSTOM_REG or cfg80211 will set it for the wiphy that called this helper.

const struct ieee80211_reg_rule * **freq_reg_info**(struct wiphy * wiphy, u32 center_freq)
get regulatory information for the given frequency

Parameters

struct wiphy * wiphy the wiphy for which we want to process this rule for

u32 center_freq Frequency in KHz for which we want regulatory information for

Description

Use this function to get the regulatory rule for a specific frequency on a given wireless device. If the device has a specific regulatory domain it wants to follow we respect that unless a country IE has been received and processed already.

Return

A valid pointer, or, when an error occurs, for example if no rule can be found, the return value is encoded using ERR_PTR(). Use IS_ERR() to check and PTR_ERR()

to obtain the numeric return value. The numeric return value will be -ERANGE if we determine the given center_freq does not even have a regulatory rule for a frequency range in the center_freq's band. See freq_in_rule_band() for our current definition of a band – this is purely subjective and right now it's 802.11 specific.

47.2.7 RFkill integration

RFkill integration in cfg80211 is almost invisible to drivers, as cfg80211 automatically registers an rfkill instance for each wireless device it knows about. Soft kill is also translated into disconnecting and turning all interfaces off, drivers are expected to turn off the device when all interfaces are down.

However, devices may have a hard RFkill line, in which case they also need to interact with the rfkill subsystem, via cfg80211. They can do this with a few helper functions documented here.

void **wiphy_rfkill_set_hw_state**(struct wiphy * wiphy, bool blocked)
notify cfg80211 about hw block state

Parameters

struct wiphy * wiphy the wiphy

bool blocked block status

void **wiphy_rfkill_start_polling**(struct wiphy * wiphy)
start polling rfkill

Parameters

struct wiphy * wiphy the wiphy

void **wiphy_rfkill_stop_polling**(struct wiphy * wiphy)
stop polling rfkill

Parameters

struct wiphy * wiphy the wiphy

47.2.8 Test mode

Test mode is a set of utility functions to allow drivers to interact with driver-specific tools to aid, for instance, factory programming.

This chapter describes how drivers interact with it, for more information see the nl80211 book's chapter on it.

struct sk_buff * **cfg80211_testmode_alloc_reply_skb**(struct wiphy * wiphy, int approxlen)
allocate testmode reply

Parameters

struct wiphy * wiphy the wiphy

int approxlen an upper bound of the length of the data that will be put into the skb

Description

This function allocates and pre-fills an skb for a reply to the testmode command. Since it is intended for a reply, calling it outside of the **testmode_cmd** operation is invalid.

The returned skb is pre-filled with the wiphy index and set up in a way that any data that is put into the skb (with skb_put(), nla_put() or similar) will end up being within the `NL80211_ATTR_TESTDATA` attribute, so all that needs to be done with the skb is adding data for the corresponding userspace tool which can then read that data out of the testdata attribute. You must not modify the skb in any other way.

When done, call `cfg80211_testmode_reply()` with the skb and return its error code as the result of the **testmode_cmd** operation.

Return

An allocated and pre-filled skb. `NULL` if any errors happen.

int **cfg80211_testmode_reply**(struct sk_buff * skb)
send the reply skb

Parameters

struct sk_buff * skb The skb, must have been allocated with `cfg80211_testmode_alloc_reply_skb()`

Description

Since calling this function will usually be the last thing before returning from the **testmode_cmd** you should return the error code. Note that this function consumes the skb regardless of the return value.

Return

An error code or 0 on success.

struct sk_buff * **cfg80211_testmode_alloc_event_skb**(struct wiphy * wiphy,
int approxlen,
gfp_t gfp)
allocate testmode event

Parameters

struct wiphy * wiphy the wiphy

int approxlen an upper bound of the length of the data that will be put into the skb

gfp_t gfp allocation flags

Description

This function allocates and pre-fills an skb for an event on the testmode multicast group.

The returned skb is set up in the same way as with `cfg80211_testmode_alloc_reply_skb()` but prepared for an event. As there, you should simply add data to it that will then end up in the `NL80211_ATTR_TESTDATA` attribute. Again, you must not modify the skb in any other way.

When done filling the skb, call `cfg80211_testmode_event()` with the skb to send the event.

Return

An allocated and pre-filled skb. `NULL` if any errors happen.

void **cfg80211_testmode_event**(struct sk_buff * skb, gfp_t gfp)
send the event

Parameters

struct sk_buff * skb The skb, must have been allocated with `cfg80211_testmode_alloc_event_skb()`

gfp_t gfp allocation flags

Description

This function sends the given **skb**, which must have been allocated by `cfg80211_testmode_alloc_event_skb()`, as an event. It always consumes it.

47.3 mac80211 subsystem (basics)

You should read and understand the information contained within this part of the book while implementing a mac80211 driver. In some chapters, advanced usage is noted, those may be skipped if this isn't needed.

This part of the book only covers station and monitor mode functionality, additional information required to implement the other modes is covered in the second part of the book.

47.3.1 Basic hardware handling

TBD

This chapter shall contain information on getting a hw struct allocated and registered with mac80211.

Since it is required to allocate rates/modes before registering a hw struct, this chapter shall also contain information on setting up the rate/mode structs.

Additionally, some discussion about the callbacks and the general programming model should be in here, including the definition of ieee80211_ops which will be referred to a lot.

Finally, a discussion of hardware capabilities should be done with references to other parts of the book.

struct **ieee80211_hw**
hardware information and state

Definition

```
struct ieee80211_hw {
  struct ieee80211_conf conf;
  struct wiphy *wiphy;
  const char *rate_control_algorithm;
  void *priv;
  unsigned long flags[BITS_TO_LONGS(NUM_IEEE80211_HW_FLAGS)];
  unsigned int extra_tx_headroom;
  unsigned int extra_beacon_tailroom;
  int vif_data_size;
  int sta_data_size;
  int chanctx_data_size;
  int txq_data_size;
  u16 queues;
  u16 max_listen_interval;
  s8 max_signal;
  u8 max_rates;
  u8 max_report_rates;
  u8 max_rate_tries;
  u16 max_rx_aggregation_subframes;
  u16 max_tx_aggregation_subframes;
  u8 max_tx_fragments;
  u8 offchannel_tx_hw_queue;
  u8 radiotap_mcs_details;
  u16 radiotap_vht_details;
  struct {
    int units_pos;
    s16 accuracy;
  } radiotap_timestamp;
  netdev_features_t netdev_features;
  u8 uapsd_queues;
  u8 uapsd_max_sp_len;
  u8 n_cipher_schemes;
  const struct ieee80211_cipher_scheme *cipher_schemes;
  u8 max_nan_de_entries;
  u8 tx_sk_pacing_shift;
  u8 weight_multiplier;
  u32 max_mtu;
};
```

Members

conf `struct ieee80211_conf`, device configuration, don't use.

wiphy This points to the `struct wiphy` allocated for this 802.11 PHY. You must fill in the **perm_addr** and **dev** members of this structure using `SET_IEEE80211_DEV()` and `SET_IEEE80211_PERM_ADDR()`. Additionally, all supported bands (with channels, bitrates) are registered here.

rate_control_algorithm rate control algorithm for this hardware. If unset (NULL), the default algorithm will be used. Must be set before calling `ieee80211_register_hw()`.

priv pointer to private area that was allocated for driver use along with this structure.

flags hardware flags, see `enum ieee80211_hw_flags`.

extra_tx_headroom headroom to reserve in each transmit skb for use by the

driver (e.g. for transmit headers.)

extra_beacon_tailroom tailroom to reserve in each beacon tx skb. Can be used by drivers to add extra IEs.

vif_data_size size (in bytes) of the drv_priv data area within `struct ieee80211_vif`.

sta_data_size size (in bytes) of the drv_priv data area within `struct ieee80211_sta`.

chanctx_data_size size (in bytes) of the drv_priv data area within `struct ieee80211_chanctx_conf`.

txq_data_size size (in bytes) of the drv_priv data area within **struct** ieee80211_txq.

queues number of available hardware transmit queues for data packets. WMM/QoS requires at least four, these queues need to have configurable access parameters.

max_listen_interval max listen interval in units of beacon interval that HW supports

max_signal Maximum value for signal (rssi) in RX information, used only when **IEEE80211_HW_SIGNAL_UNSPEC** or **IEEE80211_HW_SIGNAL_DB**

max_rates maximum number of alternate rate retry stages the hw can handle.

max_report_rates maximum number of alternate rate retry stages the hw can report back.

max_rate_tries maximum number of tries for each stage

max_rx_aggregation_subframes maximum buffer size (number of sub-frames) to be used for A-MPDU block ack receiver aggregation. This is only relevant if the device has restrictions on the number of subframes, if it relies on mac80211 to do reordering it shouldn't be set.

max_tx_aggregation_subframes maximum number of subframes in an aggregate an HT/HE device will transmit. In HT AddBA we'll advertise a constant value of 64 as some older APs crash if the window size is smaller (an example is LinkSys WRT120N with FW v1.0.07 build 002 Jun 18 2012). For AddBA to HE capable peers this value will be used.

max_tx_fragments maximum number of tx buffers per (A)-MSDU, sum of 1 + skb_shinfo(skb)->nr_frags for each skb in the frag_list.

offchannel_tx_hw_queue HW queue ID to use for offchannel TX (if `IEEE80211_HW_QUEUE_CONTROL` is set)

radiotap_mcs_details lists which MCS information can the HW reports, by default it is set to _MCS, _GI and _BW but doesn't include _FMT. Use `IEEE80211_RADIOTAP_MCS_HAVE_*` values, only adding _BW is supported today.

radiotap_vht_details lists which VHT MCS information the HW reports, the default is _GI | _BANDWIDTH. Use the `IEEE80211_RADIOTAP_VHT_KNOWN_*` values.

radiotap_timestamp Information for the radiotap timestamp field; if the **units_pos** member is set to a non-negative value then the timestamp field will be added and populated from the `struct ieee80211_rx_status` device_timestamp.

radiotap_timestamp.units_pos Must be set to a combination of a IEEE80211_RADIOTAP_TIMESTAMP_UNIT_* and a IEEE80211_RADIOTAP_TIMESTAMP_SPOS_* value.

radiotap_timestamp.accuracy If non-negative, fills the accuracy in the radiotap field and the accuracy known flag will be set.

netdev_features netdev features to be set in each netdev created from this HW. Note that not all features are usable with mac80211, other features will be rejected during HW registration.

uapsd_queues This bitmap is included in (re)association frame to indicate for each access category if it is uAPSD trigger-enabled and delivery- enabled. Use IEEE80211_WMM_IE_STA_QOSINFO_AC_* to set this bitmap. Each bit corresponds to different AC. Value ‘1’ in specific bit means that corresponding AC is both trigger- and delivery-enabled. ‘0’ means neither enabled.

uapsd_max_sp_len maximum number of total buffered frames the WMM AP may deliver to a WMM STA during any Service Period triggered by the WMM STA. Use IEEE80211_WMM_IE_STA_QOSINFO_SP_* for correct values.

n_cipher_schemes a size of an array of cipher schemes definitions.

cipher_schemes a pointer to an array of cipher scheme definitions supported by HW.

max_nan_de_entries maximum number of NAN DE functions supported by the device.

tx_sk_pacing_shift Pacing shift to set on TCP sockets when frames from them are encountered. The default should typically not be changed, unless the driver has good reasons for needing more buffers.

weight_multiplier Driver specific airtime weight multiplier used while refilling deficit of each TXQ.

max_mtu the max mtu could be set.

Description

This structure contains the configuration and hardware information for an 802.11 PHY.

enum **ieee80211_hw_flags**
hardware flags

Constants

IEEE80211_HW_HAS_RATE_CONTROL The hardware or firmware includes rate control, and cannot be controlled by the stack. As such, no rate control algorithm should be instantiated, and the TX rate reported to userspace will be taken from the TX status instead of the rate control algorithm. Note that this requires that the driver implement a number of callbacks so it has the correct information, it needs to have the **set_rts_threshold** callback and must look

at the BSS config **use_cts_prot** for G/N protection, **use_short_slot** for slot timing in 2.4 GHz and **use_short_preamble** for preambles for CCK frames.

IEEE80211_HW_RX_INCLUDES_FCS Indicates that received frames passed to the stack include the FCS at the end.

IEEE80211_HW_HOST_BROADCAST_PS_BUFFERING Some wireless LAN chipsets buffer broadcast/multicast frames for power saving stations in the hardware/firmware and others rely on the host system for such buffering. This option is used to configure the IEEE 802.11 upper layer to buffer broadcast and multicast frames when there are power saving stations so that the driver can fetch them with `ieee80211_get_buffered_bc()`.

IEEE80211_HW_SIGNAL_UNSPEC Hardware can provide signal values but we don't know its units. We expect values between 0 and **max_signal**. If possible please provide dB or dBm instead.

IEEE80211_HW_SIGNAL_DBM Hardware gives signal values in dBm, decibel difference from one milliwatt. This is the preferred method since it is standardized between different devices. **max_signal** does not need to be set.

IEEE80211_HW_NEED_DTIM_BEFORE_ASSOC This device needs to get data from beacon before association (i.e. dtim_period).

IEEE80211_HW_SPECTRUM_MGMT Hardware supports spectrum management defined in 802.11h Measurement, Channel Switch, Quieting, TPC

IEEE80211_HW_AMPDU_AGGREGATION Hardware supports 11n A-MPDU aggregation.

IEEE80211_HW_SUPPORTS_PS Hardware has power save support (i.e. can go to sleep).

IEEE80211_HW_PS_NULLFUNC_STACK Hardware requires nullfunc frame handling in stack, implies stack support for dynamic PS.

IEEE80211_HW_SUPPORTS_DYNAMIC_PS Hardware has support for dynamic PS.

IEEE80211_HW_MFP_CAPABLE Hardware supports management frame protection (MFP, IEEE 802.11w).

IEEE80211_HW_WANT_MONITOR_VIF The driver would like to be informed of a virtual monitor interface when monitor interfaces are the only active interfaces.

IEEE80211_HW_NO_AUTO_VIF The driver would like for no wlanX to be created. It is expected user-space will create vifs as desired (and thus have them named as desired).

IEEE80211_HW_SW_CRYPTO_CONTROL The driver wants to control which of the crypto algorithms can be done in software - so don't automatically try to fall back to it if hardware crypto fails, but do so only if the driver returns 1. This also forces the driver to advertise its supported cipher suites.

IEEE80211_HW_SUPPORT_FAST_XMIT The driver/hardware supports fast-xmit, this currently requires only the ability to calculate the duration for frames.

IEEE80211_HW_REPORTS_TX_ACK_STATUS Hardware can provide ack status reports of Tx frames to the stack.

IEEE80211_HW_CONNECTION_MONITOR The hardware performs its own connection monitoring, including periodic keep-alives to the AP and probing the AP on beacon loss.

IEEE80211_HW_QUEUE_CONTROL The driver wants to control per-interface queue mapping in order to use different queues (not just one per AC) for different virtual interfaces. See the doc section on HW queue control for more details.

IEEE80211_HW_SUPPORTS_PER_STA_GTK The device's crypto engine supports per-station GTKs as used by IBSS RSN or during fast transition. If the device doesn't support per-station GTKs, but can be asked not to decrypt group addressed frames, then IBSS RSN support is still possible but software crypto will be used. Advertise the wiphy flag only in that case.

IEEE80211_HW_AP_LINK_PS When operating in AP mode the device autonomously manages the PS status of connected stations. When this flag is set mac80211 will not trigger PS mode for connected stations based on the PM bit of incoming frames. Use ieee80211_start_ps()/ieee8021_end_ps() to manually configure the PS mode of connected stations.

IEEE80211_HW_TX_AMPDU_SETUP_IN_HW The device handles TX A-MPDU session setup strictly in HW. mac80211 should not attempt to do this in software.

IEEE80211_HW_SUPPORTS_RC_TABLE The driver supports using a rate selection table provided by the rate control algorithm.

IEEE80211_HW_P2P_DEV_ADDR_FOR_INTF Use the P2P Device address for any P2P Interface. This will be honoured even if more than one interface is supported.

IEEE80211_HW_TIMING_BEACON_ONLY Use sync timing from beacon frames only, to allow getting TBTT of a DTIM beacon.

IEEE80211_HW_SUPPORTS_HT_CCK_RATES Hardware supports mixing HT/CCK rates and can cope with CCK rates in an aggregation session (e.g. by not using aggregation for such frames.)

IEEE80211_HW_CHANCTX_STA_CSA Support 802.11h based channel-switch (CSA) for a single active channel while using channel contexts. When support is not enabled the default action is to disconnect when getting the CSA frame.

IEEE80211_HW_SUPPORTS_CLONED_SKBS The driver will never modify the payload or tailroom of TX skbs without copying them first.

IEEE80211_HW_SINGLE_SCAN_ON_ALL_BANDS The HW supports scanning on all bands in one command, mac80211 doesn't have to run separate scans per band.

IEEE80211_HW_TDLS_WIDER_BW The device/driver supports wider bandwidth than then BSS bandwidth for a TDLS link on the base channel.

IEEE80211_HW_SUPPORTS_AMSDU_IN_AMPDU The driver supports receiving A-MSDUs within A-MPDU.

IEEE80211_HW_BEACON_TX_STATUS The device/driver provides TX status for sent beacons.

IEEE80211_HW_NEEDS_UNIQUE_STA_ADDR Hardware (or driver) requires that each station has a unique address, i.e. each station entry can be identified by just

its MAC address; this prevents, for example, the same station from connecting to two virtual AP interfaces at the same time.

IEEE80211_HW_SUPPORTS_REORDERING_BUFFER Hardware (or driver) manages the reordering buffer internally, guaranteeing mac80211 receives frames in order and does not need to manage its own reorder buffer or BA session timeout.

IEEE80211_HW_USES_RSS The device uses RSS and thus requires parallel RX, which implies using per-CPU station statistics.

IEEE80211_HW_TX_AMSDU Hardware (or driver) supports software aggregated A-MSDU frames. Requires software tx queueing and fast-xmit support. When not using minstrel/minstrel_ht rate control, the driver must limit the maximum A-MSDU size based on the current tx rate by setting max_rc_amsdu_len in struct ieee80211_sta.

IEEE80211_HW_TX_FRAG_LIST Hardware (or driver) supports sending frag_list skbs, needed for zero-copy software A-MSDU.

IEEE80211_HW_REPORTS_LOW_ACK The driver (or firmware) reports low ack event by ieee80211_report_low_ack() based on its own algorithm. For such drivers, mac80211 packet loss mechanism will not be triggered and driver is completely depending on firmware event for station kickout.

IEEE80211_HW_SUPPORTS_TX_FRAG Hardware does fragmentation by itself. The stack will not do fragmentation. The callback for **set_frag_threshold** should be set as well.

IEEE80211_HW_SUPPORTS_TDLS_BUFFER_STA Hardware supports buffer STA on TDLS links.

IEEE80211_HW_DEAUTH_NEED_MGD_TX_PREP The driver requires the mgd_prepare_tx() callback to be called before transmission of a deauthentication frame in case the association was completed but no beacon was heard. This is required in multi-channel scenarios, where the virtual interface might not be given air time for the transmission of the frame, as it is not synced with the AP/P2P GO yet, and thus the deauthentication frame might not be transmitted.

IEEE80211_HW_DOESNT_SUPPORT_QOS_NDP The driver (or firmware) doesn't support QoS NDP for AP probing - that's most likely a driver bug.

IEEE80211_HW_BUFF_MMPDU_TXQ use the TXQ for bufferable MMPDUs, this of course requires the driver to use TXQs to start with.

IEEE80211_HW_SUPPORTS_VHT_EXT_NSS_BW (Hardware) rate control supports VHT extended NSS BW (dot11VHTExtendedNSSBWCapable). This flag will be set if the selected rate control algorithm sets `RATE_CTRL_CAPA_VHT_EXT_NSS_BW` but if the rate control is built-in then it must be set by the driver. See also the documentation for that flag.

IEEE80211_HW_STA_MMPDU_TXQ use the extra non-TID per-station TXQ for all MMPDUs on station interfaces. This of course requires the driver to use TXQs to start with.

IEEE80211_HW_TX_STATUS_NO_AMPDU_LEN Driver does not report accurate A-MPDU length in tx status information

IEEE80211_HW_SUPPORTS_MULTI_BSSID Hardware supports multi BSSID

IEEE80211_HW_SUPPORTS_ONLY_HE_MULTI_BSSID Hardware supports multi BSSID only for HE APs. Applies if **IEEE80211_HW_SUPPORTS_MULTI_BSSID** is set.

IEEE80211_HW_AMPDU_KEYBORDER_SUPPORT The card and driver is only aggregating MPDUs with the same keyid, allowing mac80211 to keep Tx A-MPDU sessions active while rekeying with Extended Key ID.

NUM_IEEE80211_HW_FLAGS number of hardware flags, used for sizing arrays

Description

These flags are used to indicate hardware capabilities to the stack. Generally, flags here should have their meaning done in a way that the simplest hardware doesn't need setting any particular flags. There are some exceptions to this rule, however, so you are advised to review these flags carefully.

void **SET_IEEE80211_DEV**(struct ieee80211_hw * hw, struct device * dev)
set device for 802.11 hardware

Parameters

struct ieee80211_hw * hw the `struct ieee80211_hw` to set the device for

struct device * dev the `struct device` of this 802.11 device

void **SET_IEEE80211_PERM_ADDR**(struct ieee80211_hw * hw, const u8 * addr)
set the permanent MAC address for 802.11 hardware

Parameters

struct ieee80211_hw * hw the `struct ieee80211_hw` to set the MAC address for

const u8 * addr the address to set

struct **ieee80211_ops**
callbacks from mac80211 to the driver

Definition

```
struct ieee80211_ops {
  void (*tx)(struct ieee80211_hw *hw,struct ieee80211_tx_control *control,␣
↪struct sk_buff *skb);
  int (*start)(struct ieee80211_hw *hw);
  void (*stop)(struct ieee80211_hw *hw);
#ifdef CONFIG_PM;
  int (*suspend)(struct ieee80211_hw *hw, struct cfg80211_wowlan *wowlan);
  int (*resume)(struct ieee80211_hw *hw);
  void (*set_wakeup)(struct ieee80211_hw *hw, bool enabled);
#endif;
  int (*add_interface)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*change_interface)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif, enum nl80211_iftype new_type, bool p2p);
  void (*remove_interface)(struct ieee80211_hw *hw, struct ieee80211_vif␣
↪*vif);
  int (*config)(struct ieee80211_hw *hw, u32 changed);
  void (*bss_info_changed)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif,struct ieee80211_bss_conf *info, u32 changed);
```

(continues on next page)

(continued from previous page)

```
  int (*start_ap)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (*stop_ap)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  u64 (*prepare_multicast)(struct ieee80211_hw *hw, struct netdev_hw_addr_
↪list *mc_list);
  void (*configure_filter)(struct ieee80211_hw *hw,unsigned int changed_
↪flags,unsigned int *total_flags, u64 multicast);
  void (*config_iface_filter)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif,unsigned int filter_flags, unsigned int changed_flags);
  int (*set_tim)(struct ieee80211_hw *hw, struct ieee80211_sta *sta, bool␣
↪set);
  int (*set_key)(struct ieee80211_hw *hw, enum set_key_cmd cmd,struct␣
↪ieee80211_vif *vif, struct ieee80211_sta *sta, struct ieee80211_key_conf␣
↪*key);
  void (*update_tkip_key)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif,struct ieee80211_key_conf *conf,struct ieee80211_sta *sta, u32 iv32,
↪ u16 *phase1key);
  void (*set_rekey_data)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪ struct cfg80211_gtk_rekey_data *data);
  void (*set_default_unicast_key)(struct ieee80211_hw *hw, struct␣
↪ieee80211_vif *vif, int idx);
  int (*hw_scan)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪struct ieee80211_scan_request *req);
  void (*cancel_hw_scan)(struct ieee80211_hw *hw, struct ieee80211_vif␣
↪*vif);
  int (*sched_scan_start)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif,struct cfg80211_sched_scan_request *req, struct ieee80211_scan_ies␣
↪*ies);
  int (*sched_scan_stop)(struct ieee80211_hw *hw, struct ieee80211_vif␣
↪*vif);
  void (*sw_scan_start)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,␣
↪const u8 *mac_addr);
  void (*sw_scan_complete)(struct ieee80211_hw *hw, struct ieee80211_vif␣
↪*vif);
  int (*get_stats)(struct ieee80211_hw *hw, struct ieee80211_low_level_
↪stats *stats);
  void (*get_key_seq)(struct ieee80211_hw *hw,struct ieee80211_key_conf␣
↪*key, struct ieee80211_key_seq *seq);
  int (*set_frag_threshold)(struct ieee80211_hw *hw, u32 value);
  int (*set_rts_threshold)(struct ieee80211_hw *hw, u32 value);
  int (*sta_add)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪struct ieee80211_sta *sta);
  int (*sta_remove)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪struct ieee80211_sta *sta);
#ifdef CONFIG_MAC80211_DEBUGFS;
  void (*sta_add_debugfs)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif,struct ieee80211_sta *sta, struct dentry *dir);
#endif;
  void (*sta_notify)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪enum sta_notify_cmd, struct ieee80211_sta *sta);
  int (*sta_set_txpwr)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,␣
↪struct ieee80211_sta *sta);
  int (*sta_state)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,
↪struct ieee80211_sta *sta,enum ieee80211_sta_state old_state, enum␣
↪ieee80211_sta_state new_state);
  void (*sta_pre_rcu_remove)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif, struct ieee80211_sta *sta);
```

(continues on next page)

(continued from previous page)

```
  void (*sta_rc_update)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪struct ieee80211_sta *sta, u32 changed);
  void (*sta_rate_tbl_update)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif, struct ieee80211_sta *sta);
  void (*sta_statistics)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪struct ieee80211_sta *sta, struct station_info *sinfo);
  int (*conf_tx)(struct ieee80211_hw *hw,struct ieee80211_vif *vif, u16 ac,
↪ const struct ieee80211_tx_queue_params *params);
  u64 (*get_tsf)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (*set_tsf)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, u64␣
↪tsf);
  void (*offset_tsf)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪s64 offset);
  void (*reset_tsf)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*tx_last_beacon)(struct ieee80211_hw *hw);
  int (*ampdu_action)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,␣
↪struct ieee80211_ampdu_params *params);
  int (*get_survey)(struct ieee80211_hw *hw, int idx, struct survey_info␣
↪*survey);
  void (*rfkill_poll)(struct ieee80211_hw *hw);
  void (*set_coverage_class)(struct ieee80211_hw *hw, s16 coverage_class);
#ifdef CONFIG_NL80211_TESTMODE;
  int (*testmode_cmd)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪void *data, int len);
  int (*testmode_dump)(struct ieee80211_hw *hw, struct sk_buff *skb,struct␣
↪netlink_callback *cb, void *data, int len);
#endif;
  void (*flush)(struct ieee80211_hw *hw, struct ieee80211_vif *vif, u32␣
↪queues, bool drop);
  void (*channel_switch)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪ struct ieee80211_channel_switch *ch_switch);
  int (*set_antenna)(struct ieee80211_hw *hw, u32 tx_ant, u32 rx_ant);
  int (*get_antenna)(struct ieee80211_hw *hw, u32 *tx_ant, u32 *rx_ant);
  int (*remain_on_channel)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif,struct ieee80211_channel *chan,int duration, enum ieee80211_roc_
↪type type);
  int (*cancel_remain_on_channel)(struct ieee80211_hw *hw, struct␣
↪ieee80211_vif *vif);
  int (*set_ringparam)(struct ieee80211_hw *hw, u32 tx, u32 rx);
  void (*get_ringparam)(struct ieee80211_hw *hw, u32 *tx, u32 *tx_max, u32␣
↪*rx, u32 *rx_max);
  bool (*tx_frames_pending)(struct ieee80211_hw *hw);
  int (*set_bitrate_mask)(struct ieee80211_hw *hw, struct ieee80211_vif␣
↪*vif, const struct cfg80211_bitrate_mask *mask);
  void (*event_callback)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪ const struct ieee80211_event *event);
  void (*allow_buffered_frames)(struct ieee80211_hw *hw,struct ieee80211_
↪sta *sta,u16 tids, int num_frames,enum ieee80211_frame_release_type␣
↪reason, bool more_data);
  void (*release_buffered_frames)(struct ieee80211_hw *hw,struct ieee80211_
↪sta *sta,u16 tids, int num_frames,enum ieee80211_frame_release_type␣
↪reason, bool more_data);
  int (*get_et_sset_count)(struct ieee80211_hw *hw, struct ieee80211_vif␣
↪*vif, int sset);
  void (*get_et_stats)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,␣
↪struct ethtool_stats *stats, u64 *data);
```

(continues on next page)

(continued from previous page)

```
  void (*get_et_strings)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪ u32 sset, u8 *data);
  void (*mgd_prepare_tx)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪ u16 duration);
  void (*mgd_protect_tdls_discover)(struct ieee80211_hw *hw, struct␣
↪ieee80211_vif *vif);
  int (*add_chanctx)(struct ieee80211_hw *hw, struct ieee80211_chanctx_
↪conf *ctx);
  void (*remove_chanctx)(struct ieee80211_hw *hw, struct ieee80211_chanctx_
↪conf *ctx);
  void (*change_chanctx)(struct ieee80211_hw *hw,struct ieee80211_chanctx_
↪conf *ctx, u32 changed);
  int (*assign_vif_chanctx)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif, struct ieee80211_chanctx_conf *ctx);
  void (*unassign_vif_chanctx)(struct ieee80211_hw *hw,struct ieee80211_
↪vif *vif, struct ieee80211_chanctx_conf *ctx);
  int (*switch_vif_chanctx)(struct ieee80211_hw *hw,struct ieee80211_vif_
↪chanctx_switch *vifs,int n_vifs, enum ieee80211_chanctx_switch_mode␣
↪mode);
  void (*reconfig_complete)(struct ieee80211_hw *hw, enum ieee80211_
↪reconfig_type reconfig_type);
#if IS_ENABLED(CONFIG_IPV6);
  void (*ipv6_addr_change)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif, struct inet6_dev *idev);
#endif;
  void (*channel_switch_beacon)(struct ieee80211_hw *hw,struct ieee80211_
↪vif *vif, struct cfg80211_chan_def *chandef);
  int (*pre_channel_switch)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif, struct ieee80211_channel_switch *ch_switch);
  int (*post_channel_switch)(struct ieee80211_hw *hw, struct ieee80211_vif␣
↪*vif);
  void (*abort_channel_switch)(struct ieee80211_hw *hw, struct ieee80211_
↪vif *vif);
  void (*channel_switch_rx_beacon)(struct ieee80211_hw *hw,struct␣
↪ieee80211_vif *vif, struct ieee80211_channel_switch *ch_switch);
  int (*join_ibss)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (*leave_ibss)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  u32 (*get_expected_throughput)(struct ieee80211_hw *hw, struct ieee80211_
↪sta *sta);
  int (*get_txpower)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪int *dbm);
  int (*tdls_channel_switch)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif,struct ieee80211_sta *sta, u8 oper_class,struct cfg80211_chan_def␣
↪*chandef, struct sk_buff *tmpl_skb, u32 ch_sw_tm_ie);
  void (*tdls_cancel_channel_switch)(struct ieee80211_hw *hw,struct␣
↪ieee80211_vif *vif, struct ieee80211_sta *sta);
  void (*tdls_recv_channel_switch)(struct ieee80211_hw *hw,struct␣
↪ieee80211_vif *vif, struct ieee80211_tdls_ch_sw_params *params);
  void (*wake_tx_queue)(struct ieee80211_hw *hw, struct ieee80211_txq␣
↪*txq);
  void (*sync_rx_queues)(struct ieee80211_hw *hw);
  int (*start_nan)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,␣
↪struct cfg80211_nan_conf *conf);
  int (*stop_nan)(struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (*nan_change_conf)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪ struct cfg80211_nan_conf *conf, u32 changes);
```

(continues on next page)

(continued from previous page)

```
  int (*add_nan_func)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,␣
↪const struct cfg80211_nan_func *nan_func);
  void (*del_nan_func)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,␣
↪u8 instance_id);
  bool (*can_aggregate_in_amsdu)(struct ieee80211_hw *hw,struct sk_buff␣
↪*head, struct sk_buff *skb);
  int (*get_ftm_responder_stats)(struct ieee80211_hw *hw,struct ieee80211_
↪vif *vif, struct cfg80211_ftm_responder_stats *ftm_stats);
  int (*start_pmsr)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪struct cfg80211_pmsr_request *request);
  void (*abort_pmsr)(struct ieee80211_hw *hw, struct ieee80211_vif *vif,␣
↪struct cfg80211_pmsr_request *request);
  int (*set_tid_config)(struct ieee80211_hw *hw,struct ieee80211_vif *vif,
↪struct ieee80211_sta *sta, struct cfg80211_tid_config *tid_conf);
  int (*reset_tid_config)(struct ieee80211_hw *hw,struct ieee80211_vif␣
↪*vif, struct ieee80211_sta *sta, u8 tids);
};
```

Members

tx Handler that 802.11 module calls for each transmitted frame. skb contains the buffer starting from the IEEE 802.11 header. The low-level driver should send the frame out based on configuration in the TX control data. This handler should, preferably, never fail and stop queues appropriately. Must be atomic.

start Called before the first netdevice attached to the hardware is enabled. This should turn on the hardware and must turn on frame reception (for possibly enabled monitor interfaces.) Returns negative error codes, these may be seen in userspace, or zero. When the device is started it should not have a MAC address to avoid acknowledging frames before a non-monitor device is added. Must be implemented and can sleep.

stop Called after last netdevice attached to the hardware is disabled. This should turn off the hardware (at least it must turn off frame reception.) May be called right after add_interface if that rejects an interface. If you added any work onto the mac80211 workqueue you should ensure to cancel it on this callback. Must be implemented and can sleep.

suspend Suspend the device; mac80211 itself will quiesce before and stop transmitting and doing any other configuration, and then ask the device to suspend. This is only invoked when WoWLAN is configured, otherwise the device is deconfigured completely and reconfigured at resume time. The driver may also impose special conditions under which it wants to use the "normal" suspend (deconfigure), say if it only supports WoWLAN when the device is associated. In this case, it must return 1 from this function.

resume If WoWLAN was configured, this indicates that mac80211 is now resuming its operation, after this the device must be fully functional again. If this returns an error, the only way out is to also unregister the device. If it returns 1, then mac80211 will also go through the regular complete restart on resume.

set_wakeup Enable or disable wakeup when WoWLAN configuration is modified. The reason is that device_set_wakeup_enable() is supposed to be called when the configuration changes, not only in suspend().

add_interface Called when a netdevice attached to the hardware is enabled. Because it is not called for monitor mode devices, **start** and **stop** must be implemented. The driver should perform any initialization it needs before the device can be enabled. The initial configuration for the interface is given in the conf parameter. The callback may refuse to add an interface by returning a negative error code (which will be seen in userspace.) Must be implemented and can sleep.

change_interface Called when a netdevice changes type. This callback is optional, but only if it is supported can interface types be switched while the interface is UP. The callback may sleep. Note that while an interface is being switched, it will not be found by the interface iteration callbacks.

remove_interface Notifies a driver that an interface is going down. The **stop** callback is called after this if it is the last interface and no monitor interfaces are present. When all interfaces are removed, the MAC address in the hardware must be cleared so the device no longer acknowledges packets, the mac_addr member of the conf structure is, however, set to the MAC address of the device going away. Hence, this callback must be implemented. It can sleep.

config Handler for configuration requests. IEEE 802.11 code calls this function to change hardware configuration, e.g., channel. This function should never fail but returns a negative error code if it does. The callback can sleep.

bss_info_changed Handler for configuration requests related to BSS parameters that may vary during BSS' s lifespan, and may affect low level driver (e.g. assoc/disassoc status, erp parameters). This function should not be used if no BSS has been set, unless for association indication. The **changed** parameter indicates which of the bss parameters has changed when a call is made. The callback can sleep.

start_ap Start operation on the AP interface, this is called after all the information in bss_conf is set and beacon can be retrieved. A channel context is bound before this is called. Note that if the driver uses software scan or ROC, this (and **stop_ap**) isn' t called when the AP is just "paused" for scanning/ROC, which is indicated by the beacon being disabled/enabled via **bss_info_changed**.

stop_ap Stop operation on the AP interface.

prepare_multicast Prepare for multicast filter configuration. This callback is optional, and its return value is passed to configure_filter(). This callback must be atomic.

configure_filter Configure the device' s RX filter. See the section "Frame filtering" for more information. This callback must be implemented and can sleep.

config_iface_filter Configure the interface's RX filter. This callback is optional and is used to configure which frames should be passed to mac80211. The filter_flags is the combination of FIF_* flags. The changed_flags is a bit mask that indicates which flags are changed. This callback can sleep.

set_tim Set TIM bit. mac80211 calls this function when a TIM bit must be set or cleared for a given STA. Must be atomic.

set_key See the section "Hardware crypto acceleration" This callback is only called between add_interface and remove_interface calls, i.e. while the given virtual interface is enabled. Returns a negative error code if the key can't be added. The callback can sleep.

update_tkip_key See the section "Hardware crypto acceleration" This callback will be called in the context of Rx. Called for drivers which set IEEE80211_KEY_FLAG_TKIP_REQ_RX_P1_KEY. The callback must be atomic.

set_rekey_data If the device supports GTK rekeying, for example while the host is suspended, it can assign this callback to retrieve the data necessary to do GTK rekeying, this is the KEK, KCK and replay counter. After rekeying was done it should (for example during resume) notify userspace of the new replay counter using ieee80211_gtk_rekey_notify().

set_default_unicast_key Set the default (unicast) key index, useful for WEP when the device sends data packets autonomously, e.g. for ARP offloading. The index can be 0-3, or -1 for unsetting it.

hw_scan Ask the hardware to service the scan request, no need to start the scan state machine in stack. The scan must honour the channel configuration done by the regulatory agent in the wiphy's registered bands. The hardware (or the driver) needs to make sure that power save is disabled. The **req** ie/ie_len members are rewritten by mac80211 to contain the entire IEs after the SSID, so that drivers need not look at these at all but just send them after the SSID – mac80211 includes the (extended) supported rates and HT information (where applicable). When the scan finishes, `ieee80211_scan_completed()` must be called; note that it also must be called when the scan cannot finish due to any error unless this callback returned a negative error code. This callback is also allowed to return the special return value 1, this indicates that hardware scan isn't desirable right now and a software scan should be done instead. A driver wishing to use this capability must ensure its (hardware) scan capabilities aren't advertised as more capable than mac80211's software scan is. The callback can sleep.

cancel_hw_scan Ask the low-level tp cancel the active hw scan. The driver should ask the hardware to cancel the scan (if possible), but the scan will be completed only after the driver will call `ieee80211_scan_completed()`. This callback is needed for wowlan, to prevent enqueueing a new scan_work after the low-level driver was already suspended. The callback can sleep.

sched_scan_start Ask the hardware to start scanning repeatedly at specific intervals. The driver must call the ieee80211_sched_scan_results() function whenever it finds results. This process will continue until sched_scan_stop is called.

sched_scan_stop Tell the hardware to stop an ongoing scheduled scan. In this case, ieee80211_sched_scan_stopped() must not be called.

sw_scan_start Notifier function that is called just before a software scan is started. Can be NULL, if the driver doesn't need this notification. The mac_addr parameter allows supporting NL80211_SCAN_FLAG_RANDOM_ADDR, the driver may set the NL80211_FEATURE_SCAN_RANDOM_MAC_ADDR flag if it can use this parameter. The callback can sleep.

sw_scan_complete Notifier function that is called just after a software scan finished. Can be NULL, if the driver doesn't need this notification. The callback can sleep.

get_stats Return low-level statistics. Returns zero if statistics are available. The callback can sleep.

get_key_seq If your device implements encryption in hardware and does IV/PN assignment then this callback should be provided to read the IV/PN for the given key from hardware. The callback must be atomic.

set_frag_threshold Configuration of fragmentation threshold. Assign this if the device does fragmentation by itself. Note that to prevent the stack from doing fragmentation IEEE80211_HW_SUPPORTS_TX_FRAG should be set as well. The callback can sleep.

set_rts_threshold Configuration of RTS threshold (if device needs it) The callback can sleep.

sta_add Notifies low level driver about addition of an associated station, AP, IBSS/WDS/mesh peer etc. This callback can sleep.

sta_remove Notifies low level driver about removal of an associated station, AP, IBSS/WDS/mesh peer etc. Note that after the callback returns it isn't safe to use the pointer, not even RCU protected; no RCU grace period is guaranteed between returning here and freeing the station. See **sta_pre_rcu_remove** if needed. This callback can sleep.

sta_add_debugfs Drivers can use this callback to add debugfs files when a station is added to mac80211's station list. This callback should be within a CONFIG_MAC80211_DEBUGFS conditional. This callback can sleep.

sta_notify Notifies low level driver about power state transition of an associated station, AP, IBSS/WDS/mesh peer etc. For a VIF operating in AP mode, this callback will not be called when the flag `IEEE80211_HW_AP_LINK_PS` is set. Must be atomic.

sta_set_txpwr Configure the station tx power. This callback set the tx power for the station. This callback can sleep.

sta_state Notifies low level driver about state transition of a station (which can be the AP, a client, IBSS/WDS/mesh peer etc.) This callback is mutually exclusive with **sta_add/sta_remove**. It must not fail for down transitions but may fail for transitions up the list of states. Also note that after the callback returns it isn't safe to use the pointer, not even RCU protected - no RCU grace period is guaranteed between returning here and freeing the station. See **sta_pre_rcu_remove** if needed. The callback can sleep.

sta_pre_rcu_remove Notify driver about station removal before RCU synchronisation. This is useful if a driver needs to have station pointers protected using RCU, it can then use this call to clear the pointers instead of waiting for an RCU grace period to elapse in **sta_state**. The callback can sleep.

sta_rc_update Notifies the driver of changes to the bitrates that can be used to transmit to the station. The changes are advertised with bits from `enum ieee80211_rate_control_changed` and the values are reflected in the station data. This callback should only be used when the driver uses hardware rate

control (IEEE80211_HW_HAS_RATE_CONTROL) since otherwise the rate control algorithm is notified directly. Must be atomic.

sta_rate_tbl_update Notifies the driver that the rate table changed. This is only used if the configured rate control algorithm actually uses the new rate table API, and is therefore optional. Must be atomic.

sta_statistics Get statistics for this station. For example with beacon filtering, the statistics kept by mac80211 might not be accurate, so let the driver pre-fill the statistics. The driver can fill most of the values (indicating which by setting the filled bitmap), but not all of them make sense - see the source for which ones are possible. Statistics that the driver doesn't fill will be filled by mac80211. The callback can sleep.

conf_tx Configure TX queue parameters (EDCF (aifs, cw_min, cw_max), bursting) for a hardware TX queue. Returns a negative error code on failure. The callback can sleep.

get_tsf Get the current TSF timer value from firmware/hardware. Currently, this is only used for IBSS mode BSSID merging and debugging. Is not a required function. The callback can sleep.

set_tsf Set the TSF timer to the specified value in the firmware/hardware. Currently, this is only used for IBSS mode debugging. Is not a required function. The callback can sleep.

offset_tsf Offset the TSF timer by the specified value in the firmware/hardware. Preferred to set_tsf as it avoids delay between calling set_tsf() and hardware getting programmed, which will show up as TSF delay. Is not a required function. The callback can sleep.

reset_tsf Reset the TSF timer and allow firmware/hardware to synchronize with other STAs in the IBSS. This is only used in IBSS mode. This function is optional if the firmware/hardware takes full care of TSF synchronization. The callback can sleep.

tx_last_beacon Determine whether the last IBSS beacon was sent by us. This is needed only for IBSS mode and the result of this function is used to determine whether to reply to Probe Requests. Returns non-zero if this device sent the last beacon. The callback can sleep.

ampdu_action Perform a certain A-MPDU action. The RA/TID combination determines the destination and TID we want the ampdu action to be performed for. The action is defined through ieee80211_ampdu_mlme_action. When the action is set to IEEE80211_AMPDU_TX_OPERATIONAL the driver may neither send aggregates containing more subframes than **buf_size** nor send aggregates in a way that lost frames would exceed the buffer size. If just limiting the aggregate size, this would be possible with a buf_size of 8:

- TX: 1.....7
- RX: 2....7 (lost frame #1)
- TX: 8..1...

which is invalid since #1 was now re-transmitted well past the buffer size of 8. Correct ways to retransmit #1 would be:

- TX: 1 or
- TX: 18 or
- TX: 81

Even 189 would be wrong since 1 could be lost again.

Returns a negative error code on failure. The driver may return IEEE80211_AMPDU_TX_START_IMMEDIATE for IEEE80211_AMPDU_TX_START if the session can start immediately.

The callback can sleep.

get_survey Return per-channel survey information

rfkill_poll Poll rfkill hardware state. If you need this, you also need to set wiphy->rfkill_poll to true before registration, and need to call wiphy_rfkill_set_hw_state() in the callback. The callback can sleep.

set_coverage_class Set slot time for given coverage class as specified in IEEE 802.11-2007 section 17.3.8.6 and modify ACK timeout accordingly; coverage class equals to -1 to enable ACK timeout estimation algorithm (dynack). To disable dynack set valid value for coverage class. This callback is not required and may sleep.

testmode_cmd Implement a cfg80211 test mode command. The passed **vif** may be NULL. The callback can sleep.

testmode_dump Implement a cfg80211 test mode dump. The callback can sleep.

flush Flush all pending frames from the hardware queue, making sure that the hardware queues are empty. The **queues** parameter is a bitmap of queues to flush, which is useful if different virtual interfaces use different hardware queues; it may also indicate all queues. If the parameter **drop** is set to true, pending frames may be dropped. Note that vif can be NULL. The callback can sleep.

channel_switch Drivers that need (or want) to offload the channel switch operation for CSAs received from the AP may implement this callback. They must then call ieee80211_chswitch_done() to indicate completion of the channel switch.

set_antenna Set antenna configuration (tx_ant, rx_ant) on the device. Parameters are bitmaps of allowed antennas to use for TX/RX. Drivers may reject TX/RX mask combinations they cannot support by returning -EINVAL (also see nl80211.h **NL80211_ATTR_WIPHY_ANTENNA_TX**).

get_antenna Get current antenna configuration from device (tx_ant, rx_ant).

remain_on_channel Starts an off-channel period on the given channel, must call back to ieee80211_ready_on_channel() when on that channel. Note that normal channel traffic is not stopped as this is intended for hw offload. Frames to transmit on the off-channel channel are transmitted normally except for the IEEE80211_TX_CTL_TX_OFFCHAN flag. When the duration (which will always be non-zero) expires, the driver must call ieee80211_remain_on_channel_expired(). Note that this callback may be called while the device is in IDLE and must be accepted in this case. This callback may sleep.

cancel_remain_on_channel Requests that an ongoing off-channel period is aborted before it expires. This callback may sleep.

set_ringparam Set tx and rx ring sizes.

get_ringparam Get tx and rx ring current and maximum sizes.

tx_frames_pending Check if there is any pending frame in the hardware queues before entering power save.

set_bitrate_mask Set a mask of rates to be used for rate control selection when transmitting a frame. Currently only legacy rates are handled. The callback can sleep.

event_callback Notify driver about any event in mac80211. See `enum ieee80211_event_type` for the different types. The callback must be atomic.

allow_buffered_frames Prepare device to allow the given number of frames to go out to the given station. The frames will be sent by mac80211 via the usual TX path after this call. The TX information for frames released will also have the `IEEE80211_TX_CTL_NO_PS_BUFFER` flag set and the last one will also have `IEEE80211_TX_STATUS_EOSP` set. In case frames from multiple TIDs are released and the driver might reorder them between the TIDs, it must set the `IEEE80211_TX_STATUS_EOSP` flag on the last frame and clear it on all others and also handle the EOSP bit in the QoS header correctly. Alternatively, it can also call the `ieee80211_sta_eosp()` function. The **tids** parameter is a bitmap and tells the driver which TIDs the frames will be on; it will at most have two bits set. This callback must be atomic.

release_buffered_frames Release buffered frames according to the given parameters. In the case where the driver buffers some frames for sleeping stations mac80211 will use this callback to tell the driver to release some frames, either for PS-poll or uAPSD. Note that if the **more_data** parameter is `false` the driver must check if there are more frames on the given TIDs, and if there are more than the frames being released then it must still set the more-data bit in the frame. If the **more_data** parameter is `true`, then of course the more-data bit must always be set. The **tids** parameter tells the driver which TIDs to release frames from, for PS-poll it will always have only a single bit set. In the case this is used for a PS-poll initiated release, the **num_frames** parameter will always be 1 so code can be shared. In this case the driver must also set `IEEE80211_TX_STATUS_EOSP` flag on the TX status (and must report TX status) so that the PS-poll period is properly ended. This is used to avoid sending multiple responses for a retried PS-poll frame. In the case this is used for uAPSD, the **num_frames** parameter may be bigger than one, but the driver may send fewer frames (it must send at least one, however). In this case it is also responsible for setting the EOSP flag in the QoS header of the frames. Also, when the service period ends, the driver must set `IEEE80211_TX_STATUS_EOSP` on the last frame in the SP. Alternatively, it may call the function `ieee80211_sta_eosp()` to inform mac80211 of the end of the SP. This callback must be atomic.

get_et_sset_count Ethtool API to get string-set count.

get_et_stats Ethtool API to get a set of u64 stats.

get_et_strings Ethtool API to get a set of strings to describe stats and perhaps

other supported types of ethtool data-sets.

mgd_prepare_tx Prepare for transmitting a management frame for association before associated. In multi-channel scenarios, a virtual interface is bound to a channel before it is associated, but as it isn't associated yet it need not necessarily be given airtime, in particular since any transmission to a P2P GO needs to be synchronized against the GO's powersave state. mac80211 will call this function before transmitting a management frame prior to having successfully associated to allow the driver to give it channel time for the transmission, to get a response and to be able to synchronize with the GO. For drivers that set `IEEE80211_HW_DEAUTH_NEED_MGD_TX_PREP`, mac80211 would also call this function before transmitting a deauthentication frame in case that no beacon was heard from the AP/P2P GO. The callback will be called before each transmission and upon return mac80211 will transmit the frame right away. If duration is greater than zero, mac80211 hints to the driver the duration for which the operation is requested. The callback is optional and can (should!) sleep.

mgd_protect_tdls_discover Protect a TDLS discovery session. After sending a TDLS discovery-request, we expect a reply to arrive on the AP's channel. We must stay on the channel (no PSM, scan, etc.), since a TDLS setup-response is a direct packet not buffered by the AP. mac80211 will call this function just before the transmission of a TDLS discovery-request. The recommended period of protection is at least 2 * (DTIM period). The callback is optional and can sleep.

add_chanctx Notifies device driver about new channel context creation. This callback may sleep.

remove_chanctx Notifies device driver about channel context destruction. This callback may sleep.

change_chanctx Notifies device driver about channel context changes that may happen when combining different virtual interfaces on the same channel context with different settings This callback may sleep.

assign_vif_chanctx Notifies device driver about channel context being bound to vif. Possible use is for hw queue remapping. This callback may sleep.

unassign_vif_chanctx Notifies device driver about channel context being unbound from vif. This callback may sleep.

switch_vif_chanctx switch a number of vifs from one chanctx to another, as specified in the list of **ieee80211_vif_chanctx_switch** passed to the driver, according to the mode defined in `ieee80211_chanctx_switch_mode`. This callback may sleep.

reconfig_complete Called after a call to ieee80211_restart_hw() and during resume, when the reconfiguration has completed. This can help the driver implement the reconfiguration step (and indicate mac80211 is ready to receive frames). This callback may sleep.

ipv6_addr_change IPv6 address assignment on the given interface changed. Currently, this is only called for managed or P2P client interfaces. This callback is optional; it must not sleep.

channel_switch_beacon Starts a channel switch to a new channel. Beacons are modified to include CSA or ECSA IEs before calling this function. The corresponding count fields in these IEs must be decremented, and when they reach 1 the driver must call ieee80211_csa_finish(). Drivers which use `ieee80211_beacon_get()` get the csa counter decremented by mac80211, but must check if it is 1 using ieee80211_csa_is_complete() after the beacon has been transmitted and then call ieee80211_csa_finish(). If the CSA count starts as zero or 1, this function will not be called, since there won't be any time to beacon before the switch anyway.

pre_channel_switch This is an optional callback that is called before a channel switch procedure is started (ie. when a STA gets a CSA or a userspace initiated channel-switch), allowing the driver to prepare for the channel switch.

post_channel_switch This is an optional callback that is called after a channel switch procedure is completed, allowing the driver to go back to a normal configuration.

abort_channel_switch This is an optional callback that is called when channel switch procedure was completed, allowing the driver to go back to a normal configuration.

channel_switch_rx_beacon This is an optional callback that is called when channel switch procedure is in progress and additional beacon with CSA IE was received, allowing driver to track changes in count.

join_ibss Join an IBSS (on an IBSS interface); this is called after all information in bss_conf is set up and the beacon can be retrieved. A channel context is bound before this is called.

leave_ibss Leave the IBSS again.

get_expected_throughput extract the expected throughput towards the specified station. The returned value is expressed in Kbps. It returns 0 if the RC algorithm does not have proper data to provide.

get_txpower get current maximum tx power (in dBm) based on configuration and hardware limits.

tdls_channel_switch Start channel-switching with a TDLS peer. The driver is responsible for continually initiating channel-switching operations and returning to the base channel for communication with the AP. The driver receives a channel-switch request template and the location of the switch-timing IE within the template as part of the invocation. The template is valid only within the call, and the driver can optionally copy the skb for further re-use.

tdls_cancel_channel_switch Stop channel-switching with a TDLS peer. Both peers must be on the base channel when the call completes.

tdls_recv_channel_switch a TDLS channel-switch related frame (request or response) has been received from a remote peer. The driver gets parameters parsed from the incoming frame and may use them to continue an ongoing channel-switch operation. In addition, a channel-switch response template is provided, together with the location of the switch-timing IE within the template. The skb can only be used within the function call.

wake_tx_queue Called when new packets have been added to the queue.

sync_rx_queues Process all pending frames in RSS queues. This is a synchronization which is needed in case driver has in its RSS queues pending frames that were received prior to the control path action currently taken (e.g. disassociation) but are not processed yet.

start_nan join an existing NAN cluster, or create a new one.

stop_nan leave the NAN cluster.

nan_change_conf change NAN configuration. The data in cfg80211_nan_conf contains full new configuration and changes specify which parameters are changed with respect to the last NAN config. The driver gets both full configuration and the changed parameters since some devices may need the full configuration while others need only the changed parameters.

add_nan_func Add a NAN function. Returns 0 on success. The data in cfg80211_nan_func must not be referenced outside the scope of this call.

del_nan_func Remove a NAN function. The driver must call ieee80211_nan_func_terminated() with NL80211_NAN_FUNC_TERM_REASON_USER_REQUEST reason code upon removal.

can_aggregate_in_amsdu Called in order to determine if HW supports aggregating two specific frames in the same A-MSDU. The relation between the skbs should be symmetric and transitive. Note that while skb is always a real frame, head may or may not be an A-MSDU.

get_ftm_responder_stats Retrieve FTM responder statistics, if available. Statistics should be cumulative, currently no way to reset is provided.

start_pmsr start peer measurement (e.g. FTM) (this call can sleep)

abort_pmsr abort peer measurement (this call can sleep)

set_tid_config Apply TID specific configurations. This callback may sleep.

reset_tid_config Reset TID specific configuration for the peer. This callback may sleep.

Description

This structure contains various callbacks that the driver may handle or, in some cases, must handle, for example to configure the hardware to a new channel or to transmit a frame.

struct ieee80211_hw * **ieee80211_alloc_hw**(size_t priv_data_len, const struct ieee80211_ops * ops)

Allocate a new hardware device

Parameters

size_t priv_data_len length of private data

const struct ieee80211_ops * ops callbacks for this device

Description

This must be called once for each hardware device. The returned pointer must be used to refer to this device when calling other functions. mac80211 allocates a

private data area for the driver pointed to by **priv** in `struct ieee80211_hw`, the size of this area is given as **priv_data_len**.

Return

A pointer to the new hardware device, or `NULL` on error.

int **ieee80211_register_hw**(struct ieee80211_hw * hw)
Register hardware device

Parameters

struct ieee80211_hw * hw the device to register as returned by `ieee80211_alloc_hw()`

Description

You must call this function before any other functions in mac80211. Note that before a hardware can be registered, you need to fill the contained wiphy's information.

Return

0 on success. An error code otherwise.

void **ieee80211_unregister_hw**(struct ieee80211_hw * hw)
Unregister a hardware device

Parameters

struct ieee80211_hw * hw the hardware to unregister

Description

This function instructs mac80211 to free allocated resources and unregister netdevices from the networking subsystem.

void **ieee80211_free_hw**(struct ieee80211_hw * hw)
free hardware descriptor

Parameters

struct ieee80211_hw * hw the hardware to free

Description

This function frees everything that was allocated, including the private data for the driver. You must call `ieee80211_unregister_hw()` before calling this function.

47.3.2 PHY configuration

TBD

This chapter should describe PHY handling including start/stop callbacks and the various structures used.

struct **ieee80211_conf**
configuration of the device

Definition

```
struct ieee80211_conf {
  u32 flags;
  int power_level, dynamic_ps_timeout;
  u16 listen_interval;
  u8 ps_dtim_period;
  u8 long_frame_max_tx_count, short_frame_max_tx_count;
  struct cfg80211_chan_def chandef;
  bool radar_enabled;
  enum ieee80211_smps_mode smps_mode;
};
```

Members

flags configuration flags defined above

power_level requested transmit power (in dBm), backward compatibility value only that is set to the minimum of all interfaces

dynamic_ps_timeout The dynamic powersave timeout (in ms), see the powersave documentation below. This variable is valid only when the CONF_PS flag is set.

listen_interval listen interval in units of beacon interval

ps_dtim_period The DTIM period of the AP we're connected to, for use in power saving. Power saving will not be enabled until a beacon has been received and the DTIM period is known.

long_frame_max_tx_count Maximum number of transmissions for a "long" frame (a frame not RTS protected), called "dot11LongRetryLimit" in 802.11, but actually means the number of transmissions not the number of retries

short_frame_max_tx_count Maximum number of transmissions for a "short" frame, called "dot11ShortRetryLimit" in 802.11, but actually means the number of transmissions not the number of retries

chandef the channel definition to tune to

radar_enabled whether radar detection is enabled

smps_mode spatial multiplexing powersave mode; note that `IEEE80211_SMPS_STATIC` is used when the device is not configured for an HT channel. Note that this is only valid if channel contexts are not used, otherwise each channel context has the number of chains listed.

Description

This struct indicates how the driver shall configure the hardware.

enum **ieee80211_conf_flags**
configuration flags

Constants

IEEE80211_CONF_MONITOR there's a monitor interface present – use this to determine for example whether to calculate timestamps for packets or not, do not use instead of filter flags!

IEEE80211_CONF_PS Enable 802.11 power save mode (managed mode only). This is the power save mode defined by IEEE 802.11-2007 section 11.2, mean-

ing that the hardware still wakes up for beacons, is able to transmit frames and receive the possible acknowledgment frames. Not to be confused with hardware specific wakeup/sleep states, driver is responsible for that. See the section “Powersave support” for more.

IEEE80211_CONF_IDLE The device is running, but idle; if the flag is set the driver should be prepared to handle configuration requests but may turn the device off as much as possible. Typically, this flag will be set when an interface is set UP but not associated or scanning, but it can also be unset in that case when monitor interfaces are active.

IEEE80211_CONF_OFFCHANNEL The device is currently not on its main operating channel.

Description

Flags to define PHY configuration options

47.3.3 Virtual interfaces

TBD

This chapter should describe virtual interface basics that are relevant to the driver (VLANs, MGMT etc are not.) It should explain the use of the add_iface/remove_iface callbacks as well as the interface configuration callbacks.

Things related to AP mode should be discussed there.

Things related to supporting multiple interfaces should be in the appropriate chapter, a BIG FAT note should be here about this though and the recommendation to allow only a single interface in STA mode at first!

struct **ieee80211_vif**
per-interface data

Definition

```
struct ieee80211_vif {
  enum nl80211_iftype type;
  struct ieee80211_bss_conf bss_conf;
  u8 addr[ETH_ALEN] ;
  bool p2p;
  bool csa_active;
  bool mu_mimo_owner;
  u8 cab_queue;
  u8 hw_queue[IEEE80211_NUM_ACS];
  struct ieee80211_txq *txq;
  struct ieee80211_chanctx_conf __rcu *chanctx_conf;
  u32 driver_flags;
#ifdef CONFIG_MAC80211_DEBUGFS;
  struct dentry *debugfs_dir;
#endif;
  bool probe_req_reg;
  bool rx_mcast_action_reg;
  bool txqs_stopped[IEEE80211_NUM_ACS];
  u8 drv_priv[] ;
};
```

Members

type type of this virtual interface

bss_conf BSS configuration for this interface, either our own or the BSS we're associated to

addr address of this interface

p2p indicates whether this AP or STA interface is a p2p interface, i.e. a GO or p2p-sta respectively

csa_active marks whether a channel switch is going on. Internally it is write-protected by sdata_lock and local->mtx so holding either is fine for read access.

mu_mimo_owner indicates interface owns MU-MIMO capability

cab_queue content-after-beacon (DTIM beacon really) queue, AP mode only

hw_queue hardware queue for each AC

txq the multicast data TX queue (if driver uses the TXQ abstraction)

chanctx_conf The channel context this interface is assigned to, or `NULL` when it is not assigned. This pointer is RCU-protected due to the TX path needing to access it; even though the netdev carrier will always be off when it is `NULL` there can still be races and packets could be processed after it switches back to `NULL`.

driver_flags flags/capabilities the driver has for this interface, these need to be set (or cleared) when the interface is added or, if supported by the driver, the interface type is changed at runtime, mac80211 will never touch this field

debugfs_dir debugfs dentry, can be used by drivers to create own per interface debug files. Note that it will be NULL for the virtual monitor interface (if that is requested.)

probe_req_reg probe requests should be reported to mac80211 for this interface.

rx_mcast_action_reg multicast Action frames should be reported to mac80211 for this interface.

txqs_stopped per AC flag to indicate that intermediate TXQs are stopped, protected by fq->lock.

drv_priv data area for driver use, will always be aligned to sizeof(void *).

Description

Data in this structure is continually present for driver use during the life of a virtual interface.

47.3.4 Receive and transmit processing

what should be here

TBD

This should describe the receive and transmit paths in mac80211/the drivers as well as transmit status handling.

Frame format

As a general rule, when frames are passed between mac80211 and the driver, they start with the IEEE 802.11 header and include the same octets that are sent over the air except for the FCS which should be calculated by the hardware.

There are, however, various exceptions to this rule for advanced features:

The first exception is for hardware encryption and decryption offload where the IV/ICV may or may not be generated in hardware.

Secondly, when the hardware handles fragmentation, the frame handed to the driver from mac80211 is the MSDU, not the MPDU.

Packet alignment

Drivers always need to pass packets that are aligned to two-byte boundaries to the stack.

Additionally, should, if possible, align the payload data in a way that guarantees that the contained IP header is aligned to a four-byte boundary. In the case of regular frames, this simply means aligning the payload to a four-byte boundary (because either the IP header is directly contained, or IV/RFC1042 headers that have a length divisible by four are in front of it). If the payload data is not properly aligned and the architecture doesn't support efficient unaligned operations, mac80211 will align the data.

With A-MSDU frames, however, the payload data address must yield two modulo four because there are 14-byte 802.3 headers within the A-MSDU frames that push the IP header further back to a multiple of four again. Thankfully, the specs were sane enough this time around to require padding each A-MSDU subframe to a length that is a multiple of four.

Padding like Atheros hardware adds which is between the 802.11 header and the payload is not supported, the driver is required to move the 802.11 header to be directly in front of the payload in that case.

Calling into mac80211 from interrupts

Only `ieee80211_tx_status_irqsafe()` and `ieee80211_rx_irqsafe()` can be called in hardware interrupt context. The low-level driver must not call any other functions in hardware interrupt context. If there is a need for such call, the low-level driver should first ACK the interrupt and perform the IEEE 802.11 code call after this, e.g. from a scheduled workqueue or even tasklet function.

NOTE: If the driver opts to use the _irqsafe() functions, it may not also use the non-IRQ-safe functions!

functions/definitions

struct **ieee80211_rx_status**
receive status

Definition

```
struct ieee80211_rx_status {
  u64 mactime;
  u64 boottime_ns;
  u32 device_timestamp;
  u32 ampdu_reference;
  u32 flag;
  u16 freq: 13, freq_offset: 1;
  u8 enc_flags;
  u8 encoding:2, bw:3, he_ru:3;
  u8 he_gi:2, he_dcm:1;
  u8 rate_idx;
  u8 nss;
  u8 rx_flags;
  u8 band;
  u8 antenna;
  s8 signal;
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
  u8 ampdu_delimiter_crc;
  u8 zero_length_psdu_type;
};
```

Members

mactime value in microseconds of the 64-bit Time Synchronization Function (TSF) timer when the first data symbol (MPDU) arrived at the hardware.

boottime_ns CLOCK_BOOTTIME timestamp the frame was received at, this is needed only for beacons and probe responses that update the scan cache.

device_timestamp arbitrary timestamp for the device, mac80211 doesn't use it but can store it and pass it back to the driver for synchronisation

ampdu_reference A-MPDU reference number, must be a different value for each A-MPDU but the same for each subframe within one A-MPDU

flag RX_FLAG_*

freq frequency the radio was tuned to when receiving this frame, in MHz This field must be set for management frames, but isn't strictly needed for data (other) frames - for those it only affects radiotap reporting.

freq_offset **freq** has a positive offset of 500Khz.

enc_flags uses bits from `enum mac80211_rx_encoding_flags`

encoding `enum mac80211_rx_encoding`

bw `enum rate_info_bw`

he_ru HE RU, from `enum nl80211_he_ru_alloc`

he_gi HE GI, from `enum nl80211_he_gi`

he_dcm HE DCM value

rate_idx index of data rate into band's supported rates or MCS index if HT or VHT is used (`RX_FLAG_HT/RX_FLAG_VHT`)

nss number of streams (VHT and HE only)

rx_flags internal RX flags for mac80211

band the active band when this frame was received

antenna antenna used

signal signal strength when receiving this frame, either in dBm, in dB or unspecified depending on the hardware capabilities flags **IEEE80211_HW_SIGNAL_***

chains bitmask of receive chains for which separate signal strength values were filled.

chain_signal per-chain signal strength, in dBm (unlike **signal**, doesn't support dB or unspecified units)

ampdu_delimiter_crc A-MPDU delimiter CRC

zero_length_psdu_type radiotap type of the 0-length PSDU

Description

The low-level driver should provide this information (the subset supported by hardware) to the 802.11 code with each received frame, in the skb's control buffer (cb).

enum **mac80211_rx_encoding_flags**
MCS & bandwidth flags

Constants

RX_ENC_FLAG_SHORTPRE Short preamble was used for this frame

RX_ENC_FLAG_SHORT_GI Short guard interval was used

RX_ENC_FLAG_HT_GF This frame was received in a HT-greenfield transmission, if the driver fills this value it should add `IEEE80211_RADIOTAP_MCS_HAVE_FMT` to **hw.radiotap_mcs_details** to advertise that fact.

RX_ENC_FLAG_STBC_MASK STBC 2 bit bitmask. 1 - Nss=1, 2 - Nss=2, 3 - Nss=3

RX_ENC_FLAG_LDPC LDPC was used

RX_ENC_FLAG_BF packet was beamformed

enum **mac80211_rx_flags**
receive flags

Constants

RX_FLAG_MMIC_ERROR Michael MIC error was reported on this frame. Use together with RX_FLAG_MMIC_STRIPPED.

RX_FLAG_DECRYPTED This frame was decrypted in hardware.

RX_FLAG_MACTIME_PLCP_START The timestamp passed in the RX status (**mactime** field) is valid and contains the time the SYNC preamble was received.

RX_FLAG_MMIC_STRIPPED the Michael MIC is stripped off this frame, verification has been done by the hardware.

RX_FLAG_IV_STRIPPED The IV and ICV are stripped from this frame. If this flag is set, the stack cannot do any replay detection hence the driver or hardware will have to do that.

RX_FLAG_FAILED_FCS_CRC Set this flag if the FCS check failed on the frame.

RX_FLAG_FAILED_PLCP_CRC Set this flag if the PCLP check failed on the frame.

RX_FLAG_MACTIME_START The timestamp passed in the RX status (**mactime** field) is valid and contains the time the first symbol of the MPDU was received. This is useful in monitor mode and for proper IBSS merging.

RX_FLAG_NO_SIGNAL_VAL The signal strength value is not present. Valid only for data frames (mainly A-MPDU)

RX_FLAG_AMPDU_DETAILS A-MPDU details are known, in particular the reference number (**ampdu_reference**) must be populated and be a distinct number for each A-MPDU

RX_FLAG_PN_VALIDATED Currently only valid for CCMP/GCMP frames, this flag indicates that the PN was verified for replay protection. Note that this flag is also currently only supported when a frame is also decrypted (ie. **RX_FLAG_DECRYPTED** must be set)

RX_FLAG_DUP_VALIDATED The driver should set this flag if it did de-duplication by itself.

RX_FLAG_AMPDU_LAST_KNOWN last subframe is known, should be set on all subframes of a single A-MPDU

RX_FLAG_AMPDU_IS_LAST this subframe is the last subframe of the A-MPDU

RX_FLAG_AMPDU_DELIM_CRC_ERROR A delimiter CRC error has been detected on this subframe

RX_FLAG_AMPDU_DELIM_CRC_KNOWN The delimiter CRC field is known (the CRC is stored in the **ampdu_delimiter_crc** field)

RX_FLAG_MACTIME_END The timestamp passed in the RX status (**mactime** field) is valid and contains the time the last symbol of the MPDU (including FCS) was received.

RX_FLAG_ONLY_MONITOR Report frame only to monitor interfaces without processing it in any regular way. This is useful if drivers offload some frames but still want to report them for sniffing purposes.

RX_FLAG_SKIP_MONITOR Process and report frame to all interfaces except monitor interfaces. This is useful if drivers offload some frames but still want to report them for sniffing purposes.

RX_FLAG_AMSDU_MORE Some drivers may prefer to report separate A-MSDU subframes instead of a one huge frame for performance reasons. All, but the last MSDU from an A-MSDU should have this flag set. E.g. if an A-MSDU has 3 frames, the first 2 must have the flag set, while the 3rd (last) one must not have this flag set. The flag is used to deal with retransmission/duplication recovery properly since A-MSDU subframes share the same sequence number. Reported subframes can be either regular MSDU or singly A-MSDUs. Subframes must not be interleaved with other frames.

RX_FLAG_RADIOTAP_VENDOR_DATA This frame contains vendor-specific radiotap data in the skb->data (before the frame) as described by the `struct ieee80211_vendor_radiotap`.

RX_FLAG_MIC_STRIPPED The mic was stripped of this packet. Decryption was done by the hardware

RX_FLAG_ALLOW_SAME_PN Allow the same PN as same packet before. This is used for AMSDU subframes which can have the same PN as the first subframe.

RX_FLAG_ICV_STRIPPED The ICV is stripped from this frame. CRC checking must be done in the hardware.

RX_FLAG_AMPDU_EOF_BIT Value of the EOF bit in the A-MPDU delimiter for this frame

RX_FLAG_AMPDU_EOF_BIT_KNOWN The EOF value is known

RX_FLAG_RADIOTAP_HE HE radiotap data is present (`struct ieee80211_radiotap_he`, mac80211 will fill in

- DATA3_DATA_MCS
- DATA3_DATA_DCM
- DATA3_CODING
- DATA5_GI
- DATA5_DATA_BW_RU_ALLOC
- DATA6_NSTS
- DATA3_STBC

from the RX info data, so leave those zeroed when building this data)

RX_FLAG_RADIOTAP_HE_MU HE MU radiotap data is present (`struct ieee80211_radiotap_he_mu`)

RX_FLAG_RADIOTAP_LSIG L-SIG radiotap data is present

RX_FLAG_NO_PSDU use the frame only for radiotap reporting, with the "0-length PSDU" field included there. The value for it is in `struct`

`ieee80211_rx_status`. Note that if this value isn't known the frame shouldn't be reported.

Description

These flags are used with the **flag** member of `struct ieee80211_rx_status`.

enum **mac80211_tx_info_flags**
flags to describe transmission information/status

Constants

IEEE80211_TX_CTL_REQ_TX_STATUS require TX status callback for this frame.

IEEE80211_TX_CTL_ASSIGN_SEQ The driver has to assign a sequence number to this frame, taking care of not overwriting the fragment number and increasing the sequence number only when the IEEE80211_TX_CTL_FIRST_FRAGMENT flag is set. mac80211 will properly assign sequence numbers to QoS-data frames but cannot do so correctly for non-QoS-data and management frames because beacons need them from that counter as well and mac80211 cannot guarantee proper sequencing. If this flag is set, the driver should instruct the hardware to assign a sequence number to the frame or assign one itself. Cf. IEEE 802.11-2007 7.1.3.4.1 paragraph 3. This flag will always be set for beacons and always be clear for frames without a sequence number field.

IEEE80211_TX_CTL_NO_ACK tell the low level not to wait for an ack

IEEE80211_TX_CTL_CLEAR_PS_FILT clear powersave filter for destination station

IEEE80211_TX_CTL_FIRST_FRAGMENT this is a first fragment of the frame

IEEE80211_TX_CTL_SEND_AFTER_DTIM send this frame after DTIM beacon

IEEE80211_TX_CTL_AMPDU this frame should be sent as part of an A-MPDU

IEEE80211_TX_CTL_INJECTED Frame was injected, internal to mac80211.

IEEE80211_TX_STAT_TX_FILTERED The frame was not transmitted because the destination STA was in powersave mode. Note that to avoid race conditions, the filter must be set by the hardware or firmware upon receiving a frame that indicates that the station went to sleep (must be done on device to filter frames already on the queue) and may only be unset after mac80211 gives the OK for that by setting the IEEE80211_TX_CTL_CLEAR_PS_FILT (see above), since only then is it guaranteed that no more frames are in the hardware queue.

IEEE80211_TX_STAT_ACK Frame was acknowledged

IEEE80211_TX_STAT_AMPDU The frame was aggregated, so status is for the whole aggregation.

IEEE80211_TX_STAT_AMPDU_NO_BACK no block ack was returned, so consider using block ack request (BAR).

IEEE80211_TX_CTL_RATE_CTRL_PROBE internal to mac80211, can be set by rate control algorithms to indicate probe rate, will be cleared for fragmented frames (except on the last fragment)

IEEE80211_TX_INTFL_OFFCHAN_TX_OK Internal to mac80211. Used to indicate that a frame can be transmitted while the queues are stopped for off-channel operation.

IEEE80211_TX_INTFL_NEED_TXPROCESSING completely internal to mac80211, used to indicate that a pending frame requires TX processing before it can be sent out.

IEEE80211_TX_INTFL_RETRIED completely internal to mac80211, used to indicate that a frame was already retried due to PS

IEEE80211_TX_INTFL_DONT_ENCRYPT completely internal to mac80211, used to indicate frame should not be encrypted

IEEE80211_TX_CTL_NO_PS_BUFFER This frame is a response to a poll frame (PS-Poll or uAPSD) or a non-bufferable MMPDU and must be sent although the station is in powersave mode.

IEEE80211_TX_CTL_MORE_FRAMES More frames will be passed to the transmit function after the current frame, this can be used by drivers to kick the DMA queue only if unset or when the queue gets full.

IEEE80211_TX_INTFL_RETRANSMISSION This frame is being retransmitted after TX status because the destination was asleep, it must not be modified again (no seqno assignment, crypto, etc.)

IEEE80211_TX_INTFL_MLME_CONN_TX This frame was transmitted by the MLME code for connection establishment, this indicates that its status should kick the MLME state machine.

IEEE80211_TX_INTFL_NL80211_FRAME_TX Frame was requested through nl80211 MLME command (internal to mac80211 to figure out whether to send TX status to user space)

IEEE80211_TX_CTL_LDPC tells the driver to use LDPC for this frame

IEEE80211_TX_CTL_STBC Enables Space-Time Block Coding (STBC) for this frame and selects the maximum number of streams that it can use.

IEEE80211_TX_CTL_TX_OFFCHAN Marks this packet to be transmitted on the off-channel channel when a remain-on-channel offload is done in hardware – normal packets still flow and are expected to be handled properly by the device.

IEEE80211_TX_INTFL_TKIP_MIC_FAILURE Marks this packet to be used for TKIP testing. It will be sent out with incorrect Michael MIC key to allow TKIP countermeasures to be tested.

IEEE80211_TX_CTL_NO_CCK_RATE This frame will be sent at non CCK rate. This flag is actually used for management frame especially for P2P frames not being sent at CCK rate in 2GHz band.

IEEE80211_TX_STATUS_EOSP This packet marks the end of service period, when its status is reported the service period ends. For frames in an SP that mac80211 transmits, it is already set; for driver frames the driver may set this flag. It is also used to do the same for PS-Poll responses.

IEEE80211_TX_CTL_USE_MINRATE This frame will be sent at lowest rate. This flag is used to send nullfunc frame at minimum rate when the nullfunc is used for connection monitoring purpose.

IEEE80211_TX_CTL_DONTFRAG Don't fragment this packet even if it would be fragmented by size (this is optional, only used for monitor injection).

IEEE80211_TX_STAT_NOACK_TRANSMITTED A frame that was marked with IEEE80211_TX_CTL_NO_ACK has been successfully transmitted without any errors (like issues specific to the driver/HW). This flag must not be set for frames that don't request no-ack behaviour with IEEE80211_TX_CTL_NO_ACK.

Description

These flags are used with the **flags** member of `ieee80211_tx_info`.

Note

If you have to add new flags to the enumeration, then don't forget to update `IEEE80211_TX_TEMPORARY_FLAGS` when necessary.

enum **mac80211_tx_control_flags**
flags to describe transmit control

Constants

IEEE80211_TX_CTRL_PORT_CTRL_PROTO this frame is a port control protocol frame (e.g. EAP)

IEEE80211_TX_CTRL_PS_RESPONSE This frame is a response to a poll frame (PS-Poll or uAPSD).

IEEE80211_TX_CTRL_RATE_INJECT This frame is injected with rate information

IEEE80211_TX_CTRL_AMSDU This frame is an A-MSDU frame

IEEE80211_TX_CTRL_FAST_XMIT This frame is going through the fast_xmit path

IEEE80211_TX_CTRL_SKIP_MPATH_LOOKUP This frame skips mesh path lookup

IEEE80211_TX_CTRL_HW_80211_ENCAP This frame uses hardware encapsulation (header conversion)

Description

These flags are used in tx_info->control.flags.

enum **mac80211_rate_control_flags**
per-rate flags set by the Rate Control algorithm.

Constants

IEEE80211_TX_RC_USE_RTS_CTS Use RTS/CTS exchange for this rate.

IEEE80211_TX_RC_USE_CTS_PROTECT CTS-to-self protection is required. This is set if the current BSS requires ERP protection.

IEEE80211_TX_RC_USE_SHORT_PREAMBLE Use short preamble.

IEEE80211_TX_RC_MCS HT rate.

IEEE80211_TX_RC_GREEN_FIELD Indicates whether this rate should be used in Greenfield mode.

IEEE80211_TX_RC_40_MHZ_WIDTH Indicates if the Channel Width should be 40 MHz.

IEEE80211_TX_RC_DUP_DATA The frame should be transmitted on both of the adjacent 20 MHz channels, if the current channel type is NL80211_CHAN_HT40MINUS or NL80211_CHAN_HT40PLUS.

IEEE80211_TX_RC_SHORT_GI Short Guard interval should be used for this rate.

IEEE80211_TX_RC_VHT_MCS VHT MCS rate, in this case the idx field is split into a higher 4 bits (Nss) and lower 4 bits (MCS number)

IEEE80211_TX_RC_80_MHZ_WIDTH Indicates 80 MHz transmission

IEEE80211_TX_RC_160_MHZ_WIDTH Indicates 160 MHz transmission (80+80 isn't supported yet)

Description

These flags are set by the Rate control algorithm for each rate during tx, in the **flags** member of struct ieee80211_tx_rate.

struct **ieee80211_tx_rate**
rate selection/status

Definition

```
struct ieee80211_tx_rate {
  s8 idx;
  u16 count:5, flags:11;
};
```

Members

idx rate index to attempt to send with

count number of tries in this rate before going to the next rate

flags rate control flags (`enum mac80211_rate_control_flags`)

Description

A value of -1 for **idx** indicates an invalid rate and, if used in an array of retry rates, that no more rates should be tried.

When used for transmit status reporting, the driver should always report the rate along with the flags it used.

`struct ieee80211_tx_info` contains an array of these structs in the control information, and it will be filled by the rate control algorithm according to what should be sent. For example, if this array contains, in the format { <idx>, <count> } the information:

```
{ 3, 2 }, { 2, 2 }, { 1, 4 }, { -1, 0 }, { -1, 0 }
```

then this means that the frame should be transmitted up to twice at rate 3, up to twice at rate 2, and up to four times at rate 1 if it doesn't get acknowledged. Say it gets acknowledged by the peer after the fifth attempt, the status information should then contain:

```
{ 3, 2 }, { 2, 2 }, { 1, 1 }, { -1, 0 } ...
```

since it was transmitted twice at rate 3, twice at rate 2 and once at rate 1 after which we received an acknowledgement.

struct **ieee80211_tx_info**
skb transmit information

Definition

```
struct ieee80211_tx_info {
  u32 flags;
  u32 band:3,ack_frame_id:13,hw_queue:4, tx_time_est:10;
  union {
    struct {
      union {
        struct {
          struct ieee80211_tx_rate rates[ IEEE80211_TX_MAX_RATES];
          s8 rts_cts_rate_idx;
          u8 use_rts:1;
          u8 use_cts_prot:1;
          u8 short_preamble:1;
          u8 skip_table:1;
        };
        unsigned long jiffies;
      };
      struct ieee80211_vif *vif;
      struct ieee80211_key_conf *hw_key;
      u32 flags;
      codel_time_t enqueue_time;
    } control;
    struct {
      u64 cookie;
    } ack;
    struct {
      struct ieee80211_tx_rate rates[IEEE80211_TX_MAX_RATES];
      s32 ack_signal;
      u8 ampdu_ack_len;
      u8 ampdu_len;
      u8 antenna;
      u16 tx_time;
      bool is_valid_ack_signal;
      void *status_driver_data[19 / sizeof(void *)];
    } status;
    struct {
      struct ieee80211_tx_rate driver_rates[ IEEE80211_TX_MAX_RATES];
      u8 pad[4];
      void *rate_driver_data[ IEEE80211_TX_INFO_RATE_DRIVER_DATA_SIZE /␣
↪sizeof(void *)];
    };
    void *driver_data[ IEEE80211_TX_INFO_DRIVER_DATA_SIZE / sizeof(void␣
↪*)];
  };
};
```

Members

flags transmit info flags, defined above

band the band to transmit on (use for checking for races)

ack_frame_id internal frame ID for TX status, used internally

hw_queue HW queue to put the frame on, skb_get_queue_mapping() gives the AC

tx_time_est TX time estimate in units of 4us, used internally

{unnamed_union} anonymous

control union part for control data

{unnamed_union} anonymous

{unnamed_struct} anonymous

control.rates TX rates array to try

control.rts_cts_rate_idx rate for RTS or CTS

control.use_rts use RTS

control.use_cts_prot use RTS/CTS

control.short_preamble use short preamble (CCK only)

control.skip_table skip externally configured rate table

control.jiffies timestamp for expiry on powersave clients

control.vif virtual interface (may be NULL)

control.hw_key key to encrypt with (may be NULL)

control.flags control flags, see `enum mac80211_tx_control_flags`

control.enqueue_time enqueue time (for iTXQs)

ack union part for pure ACK data

ack.cookie cookie for the ACK

status union part for status data

status.rates attempted rates

status.ack_signal ACK signal

status.ampdu_ack_len AMPDU ack length

status.ampdu_len AMPDU length

status.antenna (legacy, kept only for iwlegacy)

status.tx_time airtime consumed for transmission

status.is_valid_ack_signal ACK signal is valid

status.status_driver_data driver use area

{unnamed_struct} anonymous

driver_rates alias to **control.rates** to reserve space

pad padding

rate_driver_data driver use area if driver needs **control.rates**

driver_data array of driver_data pointers

Description

This structure is placed in skb->cb for three uses:

(1) mac80211 TX control - mac80211 tells the driver what to do

(2) driver internal use (if applicable)

(3) TX status information - driver tells mac80211 what happened

void **ieee80211_tx_info_clear_status**(struct ieee80211_tx_info * info)
clear TX status

Parameters

struct ieee80211_tx_info * info The `struct ieee80211_tx_info` to be cleared.

Description

When the driver passes an skb back to mac80211, it must report a number of things in TX status. This function clears everything in the TX status but the rate control information (it does clear the count since you need to fill that in anyway).

NOTE

You can only use this function if you do NOT use info->driver_data! Use info->rate_driver_data instead if you need only the less space that allows.

void **ieee80211_rx**(struct ieee80211_hw * hw, struct sk_buff * skb)
receive frame

Parameters

struct ieee80211_hw * hw the hardware this frame came in on

struct sk_buff * skb the buffer to receive, owned by mac80211 after this call

Description

Use this function to hand received frames to mac80211. The receive buffer in **skb** must start with an IEEE 802.11 header. In case of a paged **skb** is used, the driver is recommended to put the ieee80211 header of the frame on the linear part of the **skb** to avoid memory allocation and/or memcpy by the stack.

This function may not be called in IRQ context. Calls to this function for a single hardware must be synchronized against each other. Calls to this function, `ieee80211_rx_ni()` and `ieee80211_rx_irqsafe()` may not be mixed for a single hardware. Must not run concurrently with `ieee80211_tx_status()` or `ieee80211_tx_status_ni()`.

In process context use instead `ieee80211_rx_ni()`.

void **ieee80211_rx_ni**(struct ieee80211_hw * hw, struct sk_buff * skb)
receive frame (in process context)

Parameters

struct ieee80211_hw * hw the hardware this frame came in on

struct sk_buff * skb the buffer to receive, owned by mac80211 after this call

Description

Like `ieee80211_rx()` but can be called in process context (internally disables bottom halves).

Calls to this function, `ieee80211_rx()` and `ieee80211_rx_irqsafe()` may not be mixed for a single hardware. Must not run concurrently with `ieee80211_tx_status()` or `ieee80211_tx_status_ni()`.

void **ieee80211_rx_irqsafe**(struct ieee80211_hw * hw, struct sk_buff * skb)
receive frame

Parameters

struct ieee80211_hw * hw the hardware this frame came in on

struct sk_buff * skb the buffer to receive, owned by mac80211 after this call

Description

Like `ieee80211_rx()` but can be called in IRQ context (internally defers to a tasklet.)

Calls to this function, `ieee80211_rx()` or `ieee80211_rx_ni()` may not be mixed for a single hardware.Must not run concurrently with `ieee80211_tx_status()` or `ieee80211_tx_status_ni()`.

struct **ieee80211_tx_status**
extended tx status info for rate control

Definition

```
struct ieee80211_tx_status {
  struct ieee80211_sta *sta;
  struct ieee80211_tx_info *info;
  struct sk_buff *skb;
  struct rate_info *rate;
};
```

Members

sta Station that the packet was transmitted for

info Basic tx status information

skb Packet skb (can be NULL if not provided by the driver)

rate The TX rate that was used when sending the packet

void **ieee80211_tx_status**(struct ieee80211_hw * hw, struct sk_buff * skb)
transmit status callback

Parameters

struct ieee80211_hw * hw the hardware the frame was transmitted by

struct sk_buff * skb the frame that was transmitted, owned by mac80211 after this call

Description

Call this function for all transmitted frames after they have been transmitted. It is permissible to not call this function for multicast frames but this can affect statistics.

This function may not be called in IRQ context. Calls to this function for a single hardware must be synchronized against each other. Calls to this function, `ieee80211_tx_status_ni()` and `ieee80211_tx_status_irqsafe()` may not be mixed for a single hardware. Must not run concurrently with `ieee80211_rx()` or `ieee80211_rx_ni()`.

void **ieee80211_tx_status_ni**(struct ieee80211_hw * hw, struct sk_buff * skb)
transmit status callback (in process context)

Parameters

struct ieee80211_hw * hw the hardware the frame was transmitted by

struct sk_buff * skb the frame that was transmitted, owned by mac80211 after this call

Description

Like `ieee80211_tx_status()` but can be called in process context.

Calls to this function, `ieee80211_tx_status()` and `ieee80211_tx_status_irqsafe()` may not be mixed for a single hardware.

void **ieee80211_tx_status_irqsafe**(struct ieee80211_hw * hw, struct sk_buff * skb)
IRQ-safe transmit status callback

Parameters

struct ieee80211_hw * hw the hardware the frame was transmitted by

struct sk_buff * skb the frame that was transmitted, owned by mac80211 after this call

Description

Like `ieee80211_tx_status()` but can be called in IRQ context (internally defers to a tasklet.)

Calls to this function, `ieee80211_tx_status()` and `ieee80211_tx_status_ni()` may not be mixed for a single hardware.

void **ieee80211_rts_get**(struct ieee80211_hw * hw, struct ieee80211_vif * vif, const void * frame, size_t frame_len, const struct ieee80211_tx_info * frame_txctl, struct ieee80211_rts * rts)
RTS frame generation function

Parameters

struct ieee80211_hw * hw pointer obtained from `ieee80211_alloc_hw()`.

struct ieee80211_vif * vif `struct ieee80211_vif` pointer from the add_interface callback.

const void * frame pointer to the frame that is going to be protected by the RTS.

size_t frame_len the frame length (in octets).

const struct ieee80211_tx_info * frame_txctl `struct ieee80211_tx_info` of the frame.

struct ieee80211_rts * rts The buffer where to store the RTS frame.

Description

If the RTS frames are generated by the host system (i.e., not in hardware/firmware), the low-level driver uses this function to receive the next RTS frame from the 802.11 code. The low-level is responsible for calling this function before and RTS frame is needed.

__le16 **ieee80211_rts_duration**(struct ieee80211_hw * hw, struct ieee80211_vif * vif, size_t frame_len, const struct ieee80211_tx_info * frame_txctl)

Get the duration field for an RTS frame

Parameters

struct ieee80211_hw * hw pointer obtained from `ieee80211_alloc_hw()`.

struct ieee80211_vif * vif `struct ieee80211_vif` pointer from the add_interface callback.

size_t frame_len the length of the frame that is going to be protected by the RTS.

const struct ieee80211_tx_info * frame_txctl `struct ieee80211_tx_info` of the frame.

Description

If the RTS is generated in firmware, but the host system must provide the duration field, the low-level driver uses this function to receive the duration field value in little-endian byteorder.

Return

The duration.

void **ieee80211_ctstoself_get**(struct ieee80211_hw * hw, struct ieee80211_vif * vif, const void * frame, size_t frame_len, const struct ieee80211_tx_info * frame_txctl, struct ieee80211_cts * cts)

CTS-to-self frame generation function

Parameters

struct ieee80211_hw * hw pointer obtained from `ieee80211_alloc_hw()`.

struct ieee80211_vif * vif `struct ieee80211_vif` pointer from the add_interface callback.

const void * frame pointer to the frame that is going to be protected by the CTS-to-self.

size_t frame_len the frame length (in octets).

const struct ieee80211_tx_info * frame_txctl struct ieee80211_tx_info of the frame.

struct ieee80211_cts * cts The buffer where to store the CTS-to-self frame.

Description

If the CTS-to-self frames are generated by the host system (i.e., not in hardware/firmware), the low-level driver uses this function to receive the next CTS-to-self frame from the 802.11 code. The low-level is responsible for calling this function before and CTS-to-self frame is needed.

__le16 **ieee80211_ctstoself_duration**(struct ieee80211_hw * hw, struct ieee80211_vif * vif, size_t frame_len, const struct ieee80211_tx_info * frame_txctl)

Get the duration field for a CTS-to-self frame

Parameters

struct ieee80211_hw * hw pointer obtained from ieee80211_alloc_hw().

struct ieee80211_vif * vif struct ieee80211_vif pointer from the add_interface callback.

size_t frame_len the length of the frame that is going to be protected by the CTS-to-self.

const struct ieee80211_tx_info * frame_txctl struct ieee80211_tx_info of the frame.

Description

If the CTS-to-self is generated in firmware, but the host system must provide the duration field, the low-level driver uses this function to receive the duration field value in little-endian byteorder.

Return

The duration.

__le16 **ieee80211_generic_frame_duration**(struct ieee80211_hw * hw, struct ieee80211_vif * vif, enum nl80211_band band, size_t frame_len, struct ieee80211_rate * rate)

Calculate the duration field for a frame

Parameters

struct ieee80211_hw * hw pointer obtained from ieee80211_alloc_hw().

struct ieee80211_vif * vif struct ieee80211_vif pointer from the add_interface callback.

enum nl80211_band band the band to calculate the frame duration on

size_t frame_len the length of the frame.

struct ieee80211_rate * rate the rate at which the frame is going to be transmitted.

Description

Calculate the duration field of some generic frame, given its length and transmission rate (in 100kbps).

Return

The duration.

void **ieee80211_wake_queue**(struct ieee80211_hw * hw, int queue)
wake specific queue

Parameters

struct ieee80211_hw * hw pointer as obtained from `ieee80211_alloc_hw()`.

int queue queue number (counted from zero).

Description

Drivers should use this function instead of netif_wake_queue.

void **ieee80211_stop_queue**(struct ieee80211_hw * hw, int queue)
stop specific queue

Parameters

struct ieee80211_hw * hw pointer as obtained from `ieee80211_alloc_hw()`.

int queue queue number (counted from zero).

Description

Drivers should use this function instead of netif_stop_queue.

void **ieee80211_wake_queues**(struct ieee80211_hw * hw)
wake all queues

Parameters

struct ieee80211_hw * hw pointer as obtained from `ieee80211_alloc_hw()`.

Description

Drivers should use this function instead of netif_wake_queue.

void **ieee80211_stop_queues**(struct ieee80211_hw * hw)
stop all queues

Parameters

struct ieee80211_hw * hw pointer as obtained from `ieee80211_alloc_hw()`.

Description

Drivers should use this function instead of netif_stop_queue.

int **ieee80211_queue_stopped**(struct ieee80211_hw * hw, int queue)
test status of the queue

Parameters

struct ieee80211_hw * hw pointer as obtained from `ieee80211_alloc_hw()`.

int queue queue number (counted from zero).

Description

Drivers should use this function instead of netif_stop_queue.

Return

`true` if the queue is stopped. `false` otherwise.

47.3.5 Frame filtering

mac80211 requires to see many management frames for proper operation, and users may want to see many more frames when in monitor mode. However, for best CPU usage and power consumption, having as few frames as possible percolate through the stack is desirable. Hence, the hardware should filter as much as possible.

To achieve this, mac80211 uses filter flags (see below) to tell the driver's configure_filter() function which frames should be passed to mac80211 and which should be filtered out.

Before configure_filter() is invoked, the prepare_multicast() callback is invoked with the parameters **mc_count** and **mc_list** for the combined multicast address list of all virtual interfaces. It's use is optional, and it returns a u64 that is passed to configure_filter(). Additionally, configure_filter() has the arguments **changed_flags** telling which flags were changed and **total_flags** with the new flag states.

If your device has no multicast address filters your driver will need to check both the `FIF_ALLMULTI` flag and the **mc_count** parameter to see whether multicast frames should be accepted or dropped.

All unsupported flags in **total_flags** must be cleared. Hardware does not support a flag if it is incapable of _passing_ the frame to the stack. Otherwise the driver must ignore the flag, but not clear it. You must _only_ clear the flag (announce no support for the flag to mac80211) if you are not able to pass the packet type to the stack (so the hardware always filters it). So for example, you should clear **FIF_CONTROL**, if your hardware always filters control frames. If your hardware always passes control frames to the kernel and is incapable of filtering them, you do _not_ clear the **FIF_CONTROL** flag. This rule applies to all other FIF flags as well.

enum **ieee80211_filter_flags**
hardware filter flags

Constants

FIF_ALLMULTI pass all multicast frames, this is used if requested by the user or if the hardware is not capable of filtering by multicast address.

FIF_FCSFAIL pass frames with failed FCS (but you need to set the `RX_FLAG_FAILED_FCS_CRC` for them)

FIF_PLCPFAIL pass frames with failed PLCP CRC (but you need to set the `RX_FLAG_FAILED_PLCP_CRC` for them

FIF_BCN_PRBRESP_PROMISC This flag is set during scanning to indicate to the hardware that it should not filter beacons or probe responses by BSSID. Filtering them can greatly reduce the amount of processing mac80211 needs to do and the amount of CPU wakeups, so you should honour this flag if possible.

FIF_CONTROL pass control frames (except for PS Poll) addressed to this station

FIF_OTHER_BSS pass frames destined to other BSSes

FIF_PSPOLL pass PS Poll frames

FIF_PROBE_REQ pass probe request frames

FIF_MCAST_ACTION pass multicast Action frames

Description

These flags determine what the filter in hardware should be programmed to let through and what should not be passed to the stack. It is always safe to pass more frames than requested, but this has negative impact on power consumption.

47.3.6 The mac80211 workqueue

mac80211 provides its own workqueue for drivers and internal mac80211 use. The workqueue is a single threaded workqueue and can only be accessed by helpers for sanity checking. Drivers must ensure all work added onto the mac80211 workqueue should be cancelled on the driver stop() callback.

mac80211 will flushed the workqueue upon interface removal and during suspend.

All work performed on the mac80211 workqueue must not acquire the RTNL lock.

void **ieee80211_queue_work**(struct ieee80211_hw * hw, struct work_struct * work)
add work onto the mac80211 workqueue

Parameters

struct ieee80211_hw * hw the hardware struct for the interface we are adding work for

struct work_struct * work the work we want to add onto the mac80211 workqueue

Description

Drivers and mac80211 use this to add work onto the mac80211 workqueue. This helper ensures drivers are not queueing work when they should not be.

void **ieee80211_queue_delayed_work**(struct ieee80211_hw * hw, struct delayed_work * dwork, unsigned long delay)
add work onto the mac80211 workqueue

Parameters

struct ieee80211_hw * hw the hardware struct for the interface we are adding work for

struct delayed_work * dwork delayable work to queue onto the mac80211 workqueue

unsigned long delay number of jiffies to wait before queueing

Description

Drivers and mac80211 use this to queue delayed work onto the mac80211 workqueue.

47.4 mac80211 subsystem (advanced)

Information contained within this part of the book is of interest only for advanced interaction of mac80211 with drivers to exploit more hardware capabilities and improve performance.

47.4.1 LED support

Mac80211 supports various ways of blinking LEDs. Wherever possible, device LEDs should be exposed as LED class devices and hooked up to the appropriate trigger, which will then be triggered appropriately by mac80211.

const char * **ieee80211_get_tx_led_name**(struct ieee80211_hw * hw)
get name of TX LED

Parameters

struct ieee80211_hw * hw the hardware to get the LED trigger name for

Description

mac80211 creates a transmit LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or `NULL` if not configured for LEDs) of the trigger so you can automatically link the LED device.

Return

The name of the LED trigger. `NULL` if not configured for LEDs.

const char * **ieee80211_get_rx_led_name**(struct ieee80211_hw * hw)
get name of RX LED

Parameters

struct ieee80211_hw * hw the hardware to get the LED trigger name for

Description

mac80211 creates a receive LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or `NULL` if not configured for LEDs) of the trigger so you can automatically link the LED device.

Return

The name of the LED trigger. `NULL` if not configured for LEDs.

const char * **ieee80211_get_assoc_led_name**(struct ieee80211_hw * hw)
get name of association LED

Parameters

struct ieee80211_hw * hw the hardware to get the LED trigger name for

Description

mac80211 creates a association LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or NULL if not configured for LEDs) of the trigger so you can automatically link the LED device.

Return

The name of the LED trigger. NULL if not configured for LEDs.

const char * **ieee80211_get_radio_led_name**(struct ieee80211_hw * hw)
get name of radio LED

Parameters

struct ieee80211_hw * hw the hardware to get the LED trigger name for

Description

mac80211 creates a radio change LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or NULL if not configured for LEDs) of the trigger so you can automatically link the LED device.

Return

The name of the LED trigger. NULL if not configured for LEDs.

struct **ieee80211_tpt_blink**
throughput blink description

Definition

```
struct ieee80211_tpt_blink {
  int throughput;
  int blink_time;
};
```

Members

throughput throughput in Kbit/sec

blink_time blink time in milliseconds (full cycle, ie. one off + one on period)

enum **ieee80211_tpt_led_trigger_flags**
throughput trigger flags

Constants

IEEE80211_TPT_LEDTRIG_FL_RADIO enable blinking with radio

IEEE80211_TPT_LEDTRIG_FL_WORK enable blinking when working

IEEE80211_TPT_LEDTRIG_FL_CONNECTED enable blinking when at least one interface is connected in some way, including being an AP

const char * **ieee80211_create_tpt_led_trigger**(struct ieee80211_hw * hw, unsigned int flags, const struct ieee80211_tpt_blink * blink_table, unsigned int blink_table_len)

create throughput LED trigger

Parameters

struct ieee80211_hw * hw the hardware to create the trigger for

unsigned int flags trigger flags, see `enum ieee80211_tpt_led_trigger_flags`

const struct ieee80211_tpt_blink * blink_table the blink table – needs to be ordered by throughput

unsigned int blink_table_len size of the blink table

Return

`NULL` (in case of error, or if no LED triggers are configured) or the name of the new trigger.

Note

This function must be called before `ieee80211_register_hw()`.

47.4.2 Hardware crypto acceleration

mac80211 is capable of taking advantage of many hardware acceleration designs for encryption and decryption operations.

The set_key() callback in the `struct ieee80211_ops` for a given device is called to enable hardware acceleration of encryption and decryption. The callback takes a **sta** parameter that will be NULL for default keys or keys used for transmission only, or point to the station information for the peer for individual keys. Multiple transmission keys with the same key index may be used when VLANs are configured for an access point.

When transmitting, the TX control data will use the **hw_key_idx** selected by the driver by modifying the `struct ieee80211_key_conf` pointed to by the **key** parameter to the set_key() function.

The set_key() call for the `SET_KEY` command should return 0 if the key is now in use, `-EOPNOTSUPP` or `-ENOSPC` if it couldn't be added; if you return 0 then hw_key_idx must be assigned to the hardware key index, you are free to use the full u8 range.

Note that in the case that the **IEEE80211_HW_SW_CRYPTO_CONTROL** flag is set, mac80211 will not automatically fall back to software crypto if enabling hardware crypto failed. The set_key() call may also return the value 1 to permit this specific key/algorithm to be done in software.

When the cmd is `DISABLE_KEY` then it must succeed.

Note that it is permissible to not decrypt a frame even if a key for it has been uploaded to hardware, the stack will not make any decision based on whether a key has been uploaded or not but rather based on the receive flags.

The `struct ieee80211_key_conf` structure pointed to by the **key** parameter is guaranteed to be valid until another call to set_key() removes it, but it can only be used as a cookie to differentiate keys.

In TKIP some HW need to be provided a phase 1 key, for RX decryption acceleration (i.e. iwlwifi). Those drivers should provide update_tkip_key handler. The update_tkip_key() call updates the driver with the new phase 1 key. This happens every time the iv16 wraps around (every 65536 packets). The set_key() call will happen only once for each key (unless the AP did rekeying), it will not include a valid phase 1 key. The valid phase 1 key is provided by update_tkip_key only. The trigger that makes mac80211 call this handler is software decryption with wrap around of iv16.

The set_default_unicast_key() call updates the default WEP key index configured to the hardware for WEP encryption type. This is required for devices that support offload of data packets (e.g. ARP responses).

Mac80211 drivers should set the **NL80211_EXT_FEATURE_CAN_REPLACE_PTK0** flag when they are able to replace in-use PTK keys according to to following requirements: 1) They do not hand over frames decrypted with the old key to

enum **set_key_cmd**
key command

Constants

SET_KEY a key is set

DISABLE_KEY a key must be disabled

Description

Used with the set_key() callback in `struct ieee80211_ops`, this indicates whether a key is being removed or added.

struct **ieee80211_key_conf**
key information

Definition

```
struct ieee80211_key_conf {
  atomic64_t tx_pn;
  u32 cipher;
  u8 icv_len;
  u8 iv_len;
  u8 hw_key_idx;
  s8 keyidx;
  u16 flags;
  u8 keylen;
  u8 key[];
};
```

Members

tx_pn PN used for TX keys, may be used by the driver as well if it needs to do software PN assignment by itself (e.g. due to TSO)

cipher The key's cipher suite selector.

icv_len The ICV length for this key type

iv_len The IV length for this key type

hw_key_idx To be set by the driver, this is the key index the driver wants to be given when a frame is transmitted and needs to be encrypted in hardware.

keyidx the key index (0-3)

flags key flags, see `enum ieee80211_key_flags`.

keylen key material length

key key material. For ALG_TKIP the key is encoded as a 256-bit (32 byte) data block: - Temporal Encryption Key (128 bits) - Temporal Authenticator Tx MIC Key (64 bits) - Temporal Authenticator Rx MIC Key (64 bits)

Description

This key information is given by mac80211 to the driver by the set_key() callback in `struct ieee80211_ops`.

enum **ieee80211_key_flags**
key flags

Constants

IEEE80211_KEY_FLAG_GENERATE_IV_MGMT This flag should be set by the driver for a CCMP/GCMP key to indicate that is requires IV generation only for management frames (MFP).

IEEE80211_KEY_FLAG_GENERATE_IV This flag should be set by the driver to indicate that it requires IV generation for this particular key. Setting this flag does not necessarily mean that SKBs will have sufficient tailroom for ICV or MIC.

IEEE80211_KEY_FLAG_GENERATE_MMIC This flag should be set by the driver for a TKIP key if it requires Michael MIC generation in software.

IEEE80211_KEY_FLAG_PAIRWISE Set by mac80211, this flag indicates that the key is pairwise rather then a shared key.

IEEE80211_KEY_FLAG_SW_MGMT_TX This flag should be set by the driver for a CCMP/GCMP key if it requires CCMP/GCMP encryption of management frames (MFP) to be done in software.

IEEE80211_KEY_FLAG_PUT_IV_SPACE This flag should be set by the driver if space should be prepared for the IV, but the IV itself should not be generated. Do not set together with **IEEE80211_KEY_FLAG_GENERATE_IV** on the same key. Setting this flag does not necessarily mean that SKBs will have sufficient tailroom for ICV or MIC.

IEEE80211_KEY_FLAG_RX_MGMT This key will be used to decrypt received management frames. The flag can help drivers that have a hardware crypto implementation that doesn't deal with management frames properly by allowing them to not upload the keys to hardware and fall back to software crypto. Note that this flag deals only with RX, if your crypto engine can't deal with TX you can also set the `IEEE80211_KEY_FLAG_SW_MGMT_TX` flag to encrypt such frames in SW.

IEEE80211_KEY_FLAG_RESERVE_TAILROOM This flag should be set by the driver for a key to indicate that sufficient tailroom must always be reserved for ICV or MIC, even when HW encryption is enabled.

IEEE80211_KEY_FLAG_PUT_MIC_SPACE This flag should be set by the driver for a TKIP key if it only requires MIC space. Do not set together with **IEEE80211_KEY_FLAG_GENERATE_MMIC** on the same key.

IEEE80211_KEY_FLAG_NO_AUTO_TX Key needs explicit Tx activation.

IEEE80211_KEY_FLAG_GENERATE_MMIE This flag should be set by the driver for a AES_CMAC key to indicate that it requires sequence number generation only

Description

These flags are used for communication about keys between the driver and mac80211, with the **flags** parameter of `struct ieee80211_key_conf`.

void **ieee80211_get_tkip_p1k**(struct ieee80211_key_conf * keyconf, struct sk_buff * skb, u16 * p1k)
get a TKIP phase 1 key

Parameters

struct ieee80211_key_conf * keyconf the parameter passed with the set key

struct sk_buff * skb the packet to take the IV32 value from that will be encrypted with this P1K

u16 * p1k a buffer to which the key will be written, as 5 u16 values

Description

This function returns the TKIP phase 1 key for the IV32 taken from the given packet.

void **ieee80211_get_tkip_p1k_iv**(struct ieee80211_key_conf * keyconf, u32 iv32, u16 * p1k)
get a TKIP phase 1 key for IV32

Parameters

struct ieee80211_key_conf * keyconf the parameter passed with the set key

u32 iv32 IV32 to get the P1K for

u16 * p1k a buffer to which the key will be written, as 5 u16 values

Description

This function returns the TKIP phase 1 key for the given IV32.

void **ieee80211_get_tkip_p2k**(struct ieee80211_key_conf * keyconf, struct sk_buff * skb, u8 * p2k)
get a TKIP phase 2 key

Parameters

struct ieee80211_key_conf * keyconf the parameter passed with the set key

struct sk_buff * skb the packet to take the IV32/IV16 values from that will be encrypted with this key

u8 * p2k a buffer to which the key will be written, 16 bytes

Description

This function computes the TKIP RC4 key for the IV values in the packet.

47.4.3 Powersave support

mac80211 has support for various powersave implementations.

First, it can support hardware that handles all powersaving by itself, such hardware should simply set the `IEEE80211_HW_SUPPORTS_PS` hardware flag. In that case, it will be told about the desired powersave mode with the `IEEE80211_CONF_PS` flag depending on the association status. The hardware must take care of sending nullfunc frames when necessary, i.e. when entering and leaving powersave mode. The hardware is required to look at the AID in beacons and signal to the AP that it woke up when it finds traffic directed to it.

`IEEE80211_CONF_PS` flag enabled means that the powersave mode defined in IEEE 802.11-2007 section 11.2 is enabled. This is not to be confused with hardware wakeup and sleep states. Driver is responsible for waking up the hardware before issuing commands to the hardware and putting it back to sleep at appropriate times.

When PS is enabled, hardware needs to wakeup for beacons and receive the buffered multicast/broadcast frames after the beacon. Also it must be possible to send frames and receive the acknowledment frame.

Other hardware designs cannot send nullfunc frames by themselves and also need software support for parsing the TIM bitmap. This is also supported by mac80211 by combining the `IEEE80211_HW_SUPPORTS_PS` and `IEEE80211_HW_PS_NULLFUNC_STACK` flags. The hardware is of course still required to pass up beacons. The hardware is still required to handle waking up for multicast traffic; if it cannot the driver must handle that as best as it can, mac80211 is too slow to do that.

Dynamic powersave is an extension to normal powersave in which the hardware stays awake for a user-specified period of time after sending a frame so that reply frames need not be buffered and therefore delayed to the next wakeup. It's a compromise of getting good enough latency when there's data traffic and still saving significantly power in idle periods.

Dynamic powersave is simply supported by mac80211 enabling and disabling PS based on traffic. Driver needs to only set `IEEE80211_HW_SUPPORTS_PS` flag and mac80211 will handle everything automatically. Additionally, hardware having support for the dynamic PS feature may set the `IEEE80211_HW_SUPPORTS_DYNAMIC_PS` flag to indicate that it can support dynamic PS mode itself. The driver needs to look at the **dynamic_ps_timeout** hardware configuration value and use it that value whenever `IEEE80211_CONF_PS` is set. In this case mac80211 will disable dynamic PS feature in stack and will just keep `IEEE80211_CONF_PS` enabled whenever user has enabled powersave.

Driver informs U-APSD client support by enabling `IEEE80211_VIF_SUPPORTS_UAPSD` flag. The mode is configured through the uapsd parameter in conf_tx() operation. Hardware needs to send the QoS Nullfunc frames and stay awake until the service period has ended. To utilize U-APSD,

dynamic powersave is disabled for voip AC and all frames from that AC are transmitted with powersave enabled.

Note: U-APSD client mode is not yet supported with `IEEE80211_HW_PS_NULLFUNC_STACK`.

47.4.4 Beacon filter support

Some hardware have beacon filter support to reduce host cpu wakeups which will reduce system power consumption. It usually works so that the firmware creates a checksum of the beacon but omits all constantly changing elements (TSF, TIM etc). Whenever the checksum changes the beacon is forwarded to the host, otherwise it will be just dropped. That way the host will only receive beacons where some relevant information (for example ERP protection or WMM settings) have changed.

Beacon filter support is advertised with the `IEEE80211_VIF_BEACON_FILTER` interface capability. The driver needs to enable beacon filter support whenever power save is enabled, that is `IEEE80211_CONF_PS` is set. When power save is enabled, the stack will not check for beacon loss and the driver needs to notify about loss of beacons with `ieee80211_beacon_loss()`.

The time (or number of beacons missed) until the firmware notifies the driver of a beacon loss event (which in turn causes the driver to call `ieee80211_beacon_loss()`) should be configurable and will be controlled by mac80211 and the roaming algorithm in the future.

Since there may be constantly changing information elements that nothing in the software stack cares about, we will, in the future, have mac80211 tell the driver which information elements are interesting in the sense that we want to see changes in them. This will include

- a list of information element IDs
- a list of OUIs for the vendor information element

Ideally, the hardware would filter out any beacons without changes in the requested elements, but if it cannot support that it may, at the expense of some efficiency, filter out only a subset. For example, if the device doesn't support checking for OUIs it should pass up all changes in all vendor information elements.

Note that change, for the sake of simplification, also includes information elements appearing or disappearing from the beacon.

Some hardware supports an "ignore list" instead, just make sure nothing that was requested is on the ignore list, and include commonly changing information element IDs in the ignore list, for example 11 (BSS load) and the various vendor-assigned IEs with unknown contents (128, 129, 133-136, 149, 150, 155, 156, 173, 176, 178, 179, 219); for forward compatibility it could also include some currently unused IDs.

In addition to these capabilities, hardware should support notifying the host of changes in the beacon RSSI. This is relevant to implement roaming when no traffic is flowing (when traffic is flowing we see the RSSI of the received data packets). This can consist in notifying the host when the RSSI changes significantly or when

it drops below or rises above configurable thresholds. In the future these thresholds will also be configured by mac80211 (which gets them from userspace) to implement them as the roaming algorithm requires.

If the hardware cannot implement this, the driver should ask it to periodically pass beacon frames to the host so that software can do the signal strength threshold checking.

void **ieee80211_beacon_loss**(struct ieee80211_vif * vif)
inform hardware does not receive beacons

Parameters

struct ieee80211_vif * vif `struct ieee80211_vif` pointer from the add_interface callback.

Description

When beacon filtering is enabled with `IEEE80211_VIF_BEACON_FILTER` and `IEEE80211_CONF_PS` is set, the driver needs to inform whenever the hardware is not receiving beacons with this function.

47.4.5 Multiple queues and QoS support

TBD

struct **ieee80211_tx_queue_params**
transmit queue configuration

Definition

```
struct ieee80211_tx_queue_params {
  u16 txop;
  u16 cw_min;
  u16 cw_max;
  u8 aifs;
  bool acm;
  bool uapsd;
  bool mu_edca;
  struct ieee80211_he_mu_edca_param_ac_rec mu_edca_param_rec;
};
```

Members

txop maximum burst time in units of 32 usecs, 0 meaning disabled

cw_min minimum contention window [a value of the form 2^n-1 in the range 1..32767]

cw_max maximum contention window [like **cw_min**]

aifs arbitration interframe space [0..255]

acm is mandatory admission control required for the access category

uapsd is U-APSD mode enabled for the queue

mu_edca is the MU EDCA configured

mu_edca_param_rec MU EDCA Parameter Record for HE

Description

The information provided in this structure is required for QoS transmit queue configuration. Cf. IEEE 802.11 7.3.2.29.

47.4.6 Access point mode support

TBD

Some parts of the if_conf should be discussed here instead

Insert notes about VLAN interfaces with hw crypto here or in the hw crypto chapter.

support for powersaving clients

In order to implement AP and P2P GO modes, mac80211 has support for client powersaving, both "legacy" PS (PS-Poll/null data) and uAPSD. There currently is no support for sAPSD.

There is one assumption that mac80211 makes, namely that a client will not poll with PS-Poll and trigger with uAPSD at the same time. Both are supported, and both can be used by the same client, but they can't be used concurrently by the same client. This simplifies the driver code.

The first thing to keep in mind is that there is a flag for complete driver implementation: `IEEE80211_HW_AP_LINK_PS`. If this flag is set, mac80211 expects the driver to handle most of the state machine for powersaving clients and will ignore the PM bit in incoming frames. Drivers then use `ieee80211_sta_ps_transition()` to inform mac80211 of stations' powersave transitions. In this mode, mac80211 also doesn't handle PS-Poll/uAPSD.

In the mode without `IEEE80211_HW_AP_LINK_PS`, mac80211 will check the PM bit in incoming frames for client powersave transitions. When a station goes to sleep, we will stop transmitting to it. There is, however, a race condition: a station might go to sleep while there is data buffered on hardware queues. If the device has support for this it will reject frames, and the driver should give the frames back to mac80211 with the `IEEE80211_TX_STAT_TX_FILTERED` flag set which will cause mac80211 to retry the frame when the station wakes up. The driver is also notified of powersave transitions by calling its **sta_notify** callback.

When the station is asleep, it has three choices: it can wake up, it can PS-Poll, or it can possibly start a uAPSD service period. Waking up is implemented by simply transmitting all buffered (and filtered) frames to the station. This is the easiest case. When the station sends a PS-Poll or a uAPSD trigger frame, mac80211 will inform the driver of this with the **allow_buffered_frames** callback; this callback is optional. mac80211 will then transmit the frames as usual and set the `IEEE80211_TX_CTL_NO_PS_BUFFER` on each frame. The last frame in the service period (or the only response to a PS-Poll) also has `IEEE80211_TX_STATUS_EOSP` set to indicate that it ends the service period; as this frame must have TX status report it also sets `IEEE80211_TX_CTL_REQ_TX_STATUS`. When TX status is reported for this frame, the service period is marked has having ended and a new one can be started by the peer.

Additionally, non-bufferable MMPDUs can also be transmitted by mac80211 with the `IEEE80211_TX_CTL_NO_PS_BUFFER` set in them.

Another race condition can happen on some devices like iwlwifi when there are frames queued for the station and it wakes up or polls; the frames that are already queued could end up being transmitted first instead, causing reordering and/or wrong processing of the EOSP. The cause is that allowing frames to be transmitted to a certain station is out-of-band communication to the device. To allow this problem to be solved, the driver can call `ieee80211_sta_block_awake()` if frames are buffered when it is notified that the station went to sleep. When all these frames have been filtered (see above), it must call the function again to indicate that the station is no longer blocked.

If the driver buffers frames in the driver for aggregation in any way, it must use the `ieee80211_sta_set_buffered()` call when it is notified of the station going to sleep to inform mac80211 of any TIDs that have frames buffered. Note that when a station wakes up this information is reset (hence the requirement to call it when informed of the station going to sleep). Then, when a service period starts for any reason, **release_buffered_frames** is called with the number of frames to be released and which TIDs they are to come from. In this case, the driver is responsible for setting the EOSP (for uAPSD) and MORE_DATA bits in the released frames, to help the **more_data** parameter is passed to tell the driver if there is more data on other TIDs – the TIDs to release frames from are ignored since mac80211 doesn't know how many frames the buffers for those TIDs contain.

If the driver also implement GO mode, where absence periods may shorten service periods (or abort PS-Poll responses), it must filter those response frames except in the case of frames that are buffered in the driver – those must remain buffered to avoid reordering. Because it is possible that no frames are released in this case, the driver must call `ieee80211_sta_eosp()` to indicate to mac80211 that the service period ended anyway.

Finally, if frames from multiple TIDs are released from mac80211 but the driver might reorder them, it must clear & set the flags appropriately (only the last frame may have `IEEE80211_TX_STATUS_EOSP`) and also take care of the EOSP and MORE_DATA bits in the frame. The driver may also use `ieee80211_sta_eosp()` in this case.

Note that if the driver ever buffers frames other than QoS-data frames, it must take care to never send a non-QoS-data frame as the last frame in a service period, adding a QoS-nulldata frame after a non-QoS-data frame if needed.

struct sk_buff * **ieee80211_get_buffered_bc**(struct ieee80211_hw * hw,
struct ieee80211_vif * vif)

accessing buffered broadcast and multicast frames

Parameters

struct ieee80211_hw * hw pointer as obtained from `ieee80211_alloc_hw()`.

struct ieee80211_vif * vif `struct ieee80211_vif` pointer from the add_interface callback.

Description

Function for accessing buffered broadcast and multicast frames. If hardware/firmware does not implement buffering of broadcast/multicast frames when

power saving is used, 802.11 code buffers them in the host memory. The low-level driver uses this function to fetch next buffered frame. In most cases, this is used when generating beacon frame.

Return

A pointer to the next buffered skb or NULL if no more buffered frames are available.

Note

buffered frames are returned only after DTIM beacon frame was generated with `ieee80211_beacon_get()` and the low-level driver must thus call `ieee80211_beacon_get()` first. `ieee80211_get_buffered_bc()` returns NULL if the previous generated beacon was not DTIM, so the low-level driver does not need to check for DTIM beacons separately and should be able to use common code for all beacons.

struct sk_buff * **ieee80211_beacon_get**(struct ieee80211_hw * hw, struct ieee80211_vif * vif)
beacon generation function

Parameters

struct ieee80211_hw * hw pointer obtained from `ieee80211_alloc_hw()`.

struct ieee80211_vif * vif `struct ieee80211_vif` pointer from the add_interface callback.

Description

See ieee80211_beacon_get_tim().

Return

See ieee80211_beacon_get_tim().

void **ieee80211_sta_eosp**(struct ieee80211_sta * pubsta)
notify mac80211 about end of SP

Parameters

struct ieee80211_sta * pubsta the station

Description

When a device transmits frames in a way that it can't tell mac80211 in the TX status about the EOSP, it must clear the `IEEE80211_TX_STATUS_EOSP` bit and call this function instead. This applies for PS-Poll as well as uAPSD.

Note that just like with _tx_status() and _rx() drivers must not mix calls to irqsafe/non-irqsafe versions, this function must not be mixed with those either. Use the all irqsafe, or all non-irqsafe, don't mix!

NB: the _irqsafe version of this function doesn't exist, no driver needs it right now. Don't call this function if you'd need the _irqsafe version, look at the git history and restore the _irqsafe version!

enum **ieee80211_frame_release_type**
frame release reason

Constants

IEEE80211_FRAME_RELEASE_PSPOLL frame released for PS-Poll

IEEE80211_FRAME_RELEASE_UAPSD frame(s) released due to frame received on trigger-enabled AC

int **ieee80211_sta_ps_transition**(struct ieee80211_sta * sta, bool start)
PS transition for connected sta

Parameters

struct ieee80211_sta * sta currently connected sta

bool start start or stop PS

Description

When operating in AP mode with the `IEEE80211_HW_AP_LINK_PS` flag set, use this function to inform mac80211 about a connected station entering/leaving PS mode.

This function may not be called in IRQ context or with softirqs enabled.

Calls to this function for a single hardware must be synchronized against each other.

Return

0 on success. -EINVAL when the requested PS mode is already set.

int **ieee80211_sta_ps_transition_ni**(struct ieee80211_sta * sta, bool start)
PS transition for connected sta (in process context)

Parameters

struct ieee80211_sta * sta currently connected sta

bool start start or stop PS

Description

Like `ieee80211_sta_ps_transition()` but can be called in process context (internally disables bottom halves). Concurrent call restriction still applies.

Return

Like `ieee80211_sta_ps_transition()`.

void **ieee80211_sta_set_buffered**(struct ieee80211_sta * sta, u8 tid, bool buffered)
inform mac80211 about driver-buffered frames

Parameters

struct ieee80211_sta * sta `struct ieee80211_sta` pointer for the sleeping station

u8 tid the TID that has buffered frames

bool buffered indicates whether or not frames are buffered for this TID

Description

If a driver buffers frames for a powersave station instead of passing them back to mac80211 for retransmission, the station may still need to be told that there are buffered frames via the TIM bit.

This function informs mac80211 whether or not there are frames that are buffered in the driver for a given TID; mac80211 can then use this data to set the TIM bit (NOTE: This may call back into the driver's set_tim call! Beware of the locking!)

If all frames are released to the station (due to PS-poll or uAPSD) then the driver needs to inform mac80211 that there no longer are frames buffered. However, when the station wakes up mac80211 assumes that all buffered frames will be transmitted and clears this data, drivers need to make sure they inform mac80211 about all buffered frames on the sleep transition (sta_notify() with `STA_NOTIFY_SLEEP`).

Note that technically mac80211 only needs to know this per AC, not per TID, but since driver buffering will inevitably happen per TID (since it is related to aggregation) it is easier to make mac80211 map the TID to the AC as required instead of keeping track in all drivers that use this API.

void **ieee80211_sta_block_awake**(struct ieee80211_hw * hw, struct ieee80211_sta * pubsta, bool block)

block station from waking up

Parameters

struct ieee80211_hw * hw the hardware

struct ieee80211_sta * pubsta the station

bool block whether to block or unblock

Description

Some devices require that all frames that are on the queues for a specific station that went to sleep are flushed before a poll response or frames after the station woke up can be delivered to that it. Note that such frames must be rejected by the driver as filtered, with the appropriate status flag.

This function allows implementing this mode in a race-free manner.

To do this, a driver must keep track of the number of frames still enqueued for a specific station. If this number is not zero when the station goes to sleep, the driver must call this function to force mac80211 to consider the station to be asleep regardless of the station's actual state. Once the number of outstanding frames reaches zero, the driver must call this function again to unblock the station. That will cause mac80211 to be able to send ps-poll responses, and if the station queried in the meantime then frames will also be sent out as a result of this. Additionally, the driver will be notified that the station woke up some time after it is unblocked, regardless of whether the station actually woke up while blocked or not.

47.4.7 Supporting multiple virtual interfaces

TBD

Note: WDS with identical MAC address should almost always be OK

Insert notes about having multiple virtual interfaces with different MAC addresses here, note which configurations are supported by mac80211, add notes about supporting hw crypto with it.

void **ieee80211_iterate_active_interfaces**(struct ieee80211_hw * hw, u32 iter_flags, void (*iterator)(void *data, u8 *mac, struct ieee80211_vif *vif), void * data)

iterate active interfaces

Parameters

struct ieee80211_hw * hw the hardware struct of which the interfaces should be iterated over

u32 iter_flags iteration flags, see `enum ieee80211_interface_iteration_flags`

void (*)(void *data, u8 *mac, struct ieee80211_vif *vif) iterator the iterator function to call

void * data first argument of the iterator function

Description

This function iterates over the interfaces associated with a given hardware that are currently active and calls the callback for them. This function allows the iterator function to sleep, when the iterator function is atomic **ieee80211_iterate_active_interfaces_atomic** can be used. Does not iterate over a new interface during add_interface().

void **ieee80211_iterate_active_interfaces_atomic**(struct ieee80211_hw * hw, u32 iter_flags, void (*iterator)(void *data, u8 *mac, struct ieee80211_vif *vif), void * data)

iterate active interfaces

Parameters

struct ieee80211_hw * hw the hardware struct of which the interfaces should be iterated over

u32 iter_flags iteration flags, see `enum ieee80211_interface_iteration_flags`

void (*)(void *data, u8 *mac, struct ieee80211_vif *vif) iterator the iterator function to call, cannot sleep

void * data first argument of the iterator function

Description

This function iterates over the interfaces associated with a given hardware that are currently active and calls the callback for them. This function requires the iterator callback function to be atomic, if that is not desired, use **ieee80211_iterate_active_interfaces** instead. Does not iterate over a new interface during add_interface().

47.4.8 Station handling

TODO

struct **ieee80211_sta**
station table entry

Definition

```
struct ieee80211_sta {
  u32 supp_rates[NUM_NL80211_BANDS];
  u8 addr[ETH_ALEN];
  u16 aid;
  struct ieee80211_sta_ht_cap ht_cap;
  struct ieee80211_sta_vht_cap vht_cap;
  struct ieee80211_sta_he_cap he_cap;
  struct ieee80211_he_6ghz_capa he_6ghz_capa;
  u16 max_rx_aggregation_subframes;
  bool wme;
  u8 uapsd_queues;
  u8 max_sp;
  u8 rx_nss;
  enum ieee80211_sta_rx_bandwidth bandwidth;
  enum ieee80211_smps_mode smps_mode;
  struct ieee80211_sta_rates __rcu *rates;
  bool tdls;
  bool tdls_initiator;
  bool mfp;
  u8 max_amsdu_subframes;
  u16 max_amsdu_len;
  bool support_p2p_ps;
  u16 max_rc_amsdu_len;
  u16 max_tid_amsdu_len[IEEE80211_NUM_TIDS];
  struct ieee80211_sta_txpwr txpwr;
  struct ieee80211_txq *txq[IEEE80211_NUM_TIDS + 1];
  u8 drv_priv[] ;
};
```

Members

supp_rates Bitmap of supported rates (per band)

addr MAC address

aid AID we assigned to the station if we’re an AP

ht_cap HT capabilities of this STA; restricted to our own capabilities

vht_cap VHT capabilities of this STA; restricted to our own capabilities

he_cap HE capabilities of this STA

he_6ghz_capa on 6 GHz, holds the HE 6 GHz band capabilities

max_rx_aggregation_subframes maximal amount of frames in a single AMPDU that this station is allowed to transmit to us. Can be modified by driver.

wme indicates whether the STA supports QoS/WME (if local devices does, otherwise always false)

uapsd_queues bitmap of queues configured for uapsd. Only valid if wme is supported. The bits order is like in IEEE80211_WMM_IE_STA_QOSINFO_AC_*.

max_sp max Service Period. Only valid if wme is supported.

rx_nss in HT/VHT, the maximum number of spatial streams the station can receive at the moment, changed by operating mode notifications and capabilities. The value is only valid after the station moves to associated state.

bandwidth current bandwidth the station can receive with

smps_mode current SMPS mode (off, static or dynamic)

rates rate control selection table

tdls indicates whether the STA is a TDLS peer

tdls_initiator indicates the STA is an initiator of the TDLS link. Only valid if the STA is a TDLS peer in the first place.

mfp indicates whether the STA uses management frame protection or not.

max_amsdu_subframes indicates the maximal number of MSDUs in a single A-MSDU. Taken from the Extended Capabilities element. 0 means unlimited.

max_amsdu_len indicates the maximal length of an A-MSDU in bytes. This field is always valid for packets with a VHT preamble. For packets with a HT preamble, additional limits apply:

- If the skb is transmitted as part of a BA agreement, the A-MSDU maximal size is min(max_amsdu_len, 4065) bytes.
- If the skb is not part of a BA agreement, the A-MSDU maximal size is min(max_amsdu_len, 7935) bytes.

Both additional HT limits must be enforced by the low level driver. This is defined by the spec (IEEE 802.11-2012 section 8.3.2.2 NOTE 2).

support_p2p_ps indicates whether the STA supports P2P PS mechanism or not.

max_rc_amsdu_len Maximum A-MSDU size in bytes recommended by rate control.

max_tid_amsdu_len Maximum A-MSDU size in bytes for this TID

txpwr the station tx power configuration

txq per-TID data TX queues (if driver uses the TXQ abstraction); note that the last entry (`IEEE80211_NUM_TIDS`) is used for non-data frames

drv_priv data area for driver use, will always be aligned to sizeof(void *), size is determined in hw information.

Description

A station table entry represents a station we are possibly communicating with. Since stations are RCU-managed in mac80211, any ieee80211_sta pointer you get access to must either be protected by rcu_read_lock() explicitly or implicitly, or

you must take good care to not use such a pointer after a call to your sta_remove callback that removed it.

enum **sta_notify_cmd**
sta notify command

Constants

STA_NOTIFY_SLEEP a station is now sleeping

STA_NOTIFY_AWAKE a sleeping station woke up

Description

Used with the sta_notify() callback in `struct ieee80211_ops`, this indicates if an associated station made a power state transition.

struct ieee80211_sta * **ieee80211_find_sta**(struct ieee80211_vif * vif, const u8 * addr)
find a station

Parameters

struct ieee80211_vif * vif virtual interface to look for station on

const u8 * addr station's address

Return

The station, if found. `NULL` otherwise.

Note

This function must be called under RCU lock and the resulting pointer is only valid under RCU lock as well.

struct ieee80211_sta * **ieee80211_find_sta_by_ifaddr**(struct ieee80211_hw * hw, const u8 * addr, const u8 * localaddr)
find a station on hardware

Parameters

struct ieee80211_hw * hw pointer as obtained from `ieee80211_alloc_hw()`

const u8 * addr remote station's address

const u8 * localaddr local address (vif->sdata->vif.addr). Use NULL for 'any'.

Return

The station, if found. `NULL` otherwise.

Note

This function must be called under RCU lock and the resulting pointer is only valid under RCU lock as well.

NOTE

You may pass NULL for localaddr, but then you will just get the first STA that matches the remote address 'addr'. We can have multiple STA associated with multiple logical stations (e.g. consider a station connecting to another BSSID on the same AP hardware without disconnecting first). In this case, the result of this method with localaddr NULL is not reliable.

Description

DO NOT USE THIS FUNCTION with localaddr NULL if at all possible.

47.4.9 Hardware scan offload

TBD

void **ieee80211_scan_completed**(struct ieee80211_hw * hw, struct cfg80211_scan_info * info)

completed hardware scan

Parameters

struct ieee80211_hw * hw the hardware that finished the scan

struct cfg80211_scan_info * info information about the completed scan

Description

When hardware scan offload is used (i.e. the hw_scan() callback is assigned) this function needs to be called by the driver to notify mac80211 that the scan finished. This function can be called from any context, including hardirq context.

47.4.10 Aggregation

TX A-MPDU aggregation

Aggregation on the TX side requires setting the hardware flag `IEEE80211_HW_AMPDU_AGGREGATION`. The driver will then be handed packets with a flag indicating A-MPDU aggregation. The driver or device is responsible for actually aggregating the frames, as well as deciding how many and which to aggregate.

When TX aggregation is started by some subsystem (usually the rate control algorithm would be appropriate) by calling the `ieee80211_start_tx_ba_session()` function, the driver will be notified via its **ampdu_action** function, with the `IEEE80211_AMPDU_TX_START` action.

In response to that, the driver is later required to call the `ieee80211_start_tx_ba_cb_irqsafe()` function, which will really start the aggregation session after the peer has also responded. If the peer responds negatively, the session will be stopped again right away. Note that it is possible for the aggregation session to be stopped before the driver has indicated that it is done setting it up, in which case it must not indicate the setup completion.

Also note that, since we also need to wait for a response from the peer, the driver is notified of the completion of the handshake by the `IEEE80211_AMPDU_TX_OPERATIONAL` action to the **ampdu_action** callback.

Similarly, when the aggregation session is stopped by the peer or something calling `ieee80211_stop_tx_ba_session()`, the driver's **ampdu_action** function will be called with the action `IEEE80211_AMPDU_TX_STOP`. In this case, the call must not fail, and the driver must later call `ieee80211_stop_tx_ba_cb_irqsafe()`. Note that the sta can get destroyed before the BA tear down is complete.

RX A-MPDU aggregation

Aggregation on the RX side requires only implementing the **ampdu_action** callback that is invoked to start/stop any block-ack sessions for RX aggregation.

When RX aggregation is started by the peer, the driver is notified via **ampdu_action** function, with the `IEEE80211_AMPDU_RX_START` action, and may reject the request in which case a negative response is sent to the peer, if it accepts it a positive response is sent.

While the session is active, the device/driver are required to de-aggregate frames and pass them up one by one to mac80211, which will handle the reorder buffer.

When the aggregation session is stopped again by the peer or ourselves, the driver's **ampdu_action** function will be called with the action `IEEE80211_AMPDU_RX_STOP`. In this case, the call must not fail.

enum **ieee80211_ampdu_mlme_action**
A-MPDU actions

Constants

IEEE80211_AMPDU_RX_START start RX aggregation

IEEE80211_AMPDU_RX_STOP stop RX aggregation

IEEE80211_AMPDU_TX_START start TX aggregation, the driver must either call `ieee80211_start_tx_ba_cb_irqsafe()` or call `ieee80211_start_tx_ba_cb_irqsafe()` with status `IEEE80211_AMPDU_TX_START_DELAY_ADDBA` to delay addba after ieee80211_start_tx_ba_cb_irqsafe is called, or just return the special status `IEEE80211_AMPDU_TX_START_IMMEDIATE`.

IEEE80211_AMPDU_TX_STOP_CONT stop TX aggregation but continue transmitting queued packets, now unaggregated. After all packets are transmitted the driver has to call `ieee80211_stop_tx_ba_cb_irqsafe()`.

IEEE80211_AMPDU_TX_STOP_FLUSH stop TX aggregation and flush all packets, called when the station is removed. There's no need or reason to call `ieee80211_stop_tx_ba_cb_irqsafe()` in this case as mac80211 assumes the session is gone and removes the station.

IEEE80211_AMPDU_TX_STOP_FLUSH_CONT called when TX aggregation is stopped but the driver hasn't called `ieee80211_stop_tx_ba_cb_irqsafe()` yet and now the connection is dropped and the station will be removed. Drivers should clean up and drop remaining packets when this is called.

IEEE80211_AMPDU_TX_OPERATIONAL TX aggregation has become operational

Description

These flags are used with the ampdu_action() callback in `struct ieee80211_ops` to indicate which action is needed.

Note that drivers MUST be able to deal with a TX aggregation session being stopped even before they OK'ed starting it by calling ieee80211_start_tx_ba_cb_irqsafe, because the peer might receive the addBA frame and send a delBA right away!

47.4.11 Spatial Multiplexing Powersave (SMPS)

SMPS (Spatial multiplexing power save) is a mechanism to conserve power in an 802.11n implementation. For details on the mechanism and rationale, please refer to 802.11 (as amended by 802.11n-2009) "11.2.3 SM power save".

The mac80211 implementation is capable of sending action frames to update the AP about the station's SMPS mode, and will instruct the driver to enter the specific mode. It will also announce the requested SMPS mode during the association handshake. Hardware support for this feature is required, and can be indicated by hardware flags.

The default mode will be "automatic", which nl80211/cfg80211 defines to be dynamic SMPS in (regular) powersave, and SMPS turned off otherwise.

To support this feature, the driver must set the appropriate hardware support flags, and handle the SMPS flag to the config() operation. It will then with this mechanism be instructed to enter the requested SMPS mode while associated to an HT AP.

void **ieee80211_request_smps**(struct ieee80211_vif * vif, enum ieee80211_smps_mode smps_mode)
request SM PS transition

Parameters

struct ieee80211_vif * vif `struct ieee80211_vif` pointer from the add_interface callback.

enum ieee80211_smps_mode smps_mode new SM PS mode

Description

This allows the driver to request an SM PS transition in managed mode. This is useful when the driver has more information than the stack about possible interference, for example by bluetooth.

enum **ieee80211_smps_mode**
spatial multiplexing power save mode

Constants

IEEE80211_SMPS_AUTOMATIC automatic

IEEE80211_SMPS_OFF off

IEEE80211_SMPS_STATIC static

IEEE80211_SMPS_DYNAMIC dynamic

IEEE80211_SMPS_NUM_MODES internal, don't use

TBD

This part of the book describes the rate control algorithm interface and how it relates to mac80211 and drivers.

47.4.12 Rate Control API

TBD

int **ieee80211_start_tx_ba_session**(struct ieee80211_sta * sta, u16 tid, u16 timeout)
Start a tx Block Ack session.

Parameters

struct ieee80211_sta * sta the station for which to start a BA session

u16 tid the TID to BA on.

u16 timeout session timeout value (in TUs)

Return

success if addBA request was sent, failure otherwise

Description

Although mac80211/low level driver/user space application can estimate the need to start aggregation on a certain RA/TID, the session level will be managed by the mac80211.

void **ieee80211_start_tx_ba_cb_irqsafe**(struct ieee80211_vif * vif, const u8 * ra, u16 tid)
low level driver ready to aggregate.

Parameters

struct ieee80211_vif * vif `struct ieee80211_vif` pointer from the add_interface callback

const u8 * ra receiver address of the BA session recipient.

u16 tid the TID to BA on.

Description

This function must be called by low level driver once it has finished with preparations for the BA session. It can be called from any context.

int **ieee80211_stop_tx_ba_session**(struct ieee80211_sta * sta, u16 tid)
Stop a Block Ack session.

Parameters

struct ieee80211_sta * sta the station whose BA session to stop

u16 tid the TID to stop BA.

Return

negative error if the TID is invalid, or no aggregation active

Description

Although mac80211/low level driver/user space application can estimate the need to stop aggregation on a certain RA/TID, the session level will be managed by the mac80211.

void **ieee80211_stop_tx_ba_cb_irqsafe**(struct ieee80211_vif * vif, const u8 * ra, u16 tid)
low level driver ready to stop aggregate.

Parameters

struct ieee80211_vif * vif struct ieee80211_vif pointer from the add_interface callback

const u8 * ra receiver address of the BA session recipient.

u16 tid the desired TID to BA on.

Description

This function must be called by low level driver once it has finished with preparations for the BA session tear down. It can be called from any context.

enum **ieee80211_rate_control_changed**
flags to indicate what changed

Constants

IEEE80211_RC_BW_CHANGED The bandwidth that can be used to transmit to this station changed. The actual bandwidth is in the station information – for HT20/40 the IEEE80211_HT_CAP_SUP_WIDTH_20_40 flag changes, for HT and VHT the bandwidth field changes.

IEEE80211_RC_SMPS_CHANGED The SMPS state of the station changed.

IEEE80211_RC_SUPP_RATES_CHANGED The supported rate set of this peer changed (in IBSS mode) due to discovering more information about the peer.

IEEE80211_RC_NSS_CHANGED N_SS (number of spatial streams) was changed by the peer

struct **ieee80211_tx_rate_control**
rate control information for/from RC algo

Definition

```
struct ieee80211_tx_rate_control {
  struct ieee80211_hw *hw;
  struct ieee80211_supported_band *sband;
  struct ieee80211_bss_conf *bss_conf;
  struct sk_buff *skb;
  struct ieee80211_tx_rate reported_rate;
  bool rts, short_preamble;
  u32 rate_idx_mask;
  u8 *rate_idx_mcs_mask;
  bool bss;
};
```

Members

hw The hardware the algorithm is invoked for.

sband The band this frame is being transmitted on.

bss_conf the current BSS configuration

skb the skb that will be transmitted, the control information in it needs to be filled in

reported_rate The rate control algorithm can fill this in to indicate which rate should be reported to userspace as the current rate and used for rate calculations in the mesh network.

rts whether RTS will be used for this frame because it is longer than the RTS threshold

short_preamble whether mac80211 will request short-preamble transmission if the selected rate supports it

rate_idx_mask user-requested (legacy) rate mask

rate_idx_mcs_mask user-requested MCS rate mask (NULL if not in use)

bss whether this frame is sent out in AP or IBSS mode

TBD

This part of the book describes mac80211 internals.

47.4.13 Key handling

Key handling basics

Key handling in mac80211 is done based on per-interface (sub_if_data) keys and per-station keys. Since each station belongs to an interface, each station key also belongs to that interface.

Hardware acceleration is done on a best-effort basis for algorithms that are implemented in software, for each key the hardware is asked to enable that key for offloading but if it cannot do that the key is simply kept for software encryption (unless it is for an algorithm that isn't implemented in software). There is currently no way of knowing whether a key is handled in SW or HW except by looking into debugfs.

All key management is internally protected by a mutex. Within all other parts of mac80211, key references are, just as STA structure references, protected by RCU. Note, however, that some things are unprotected, namely the key->sta dereferences within the hardware acceleration functions. This means that sta_info_destroy() must remove the key which waits for an RCU grace period.

MORE TBD

TBD

47.4.14 Receive processing

TBD

47.4.15 Transmit processing

TBD

47.4.16 Station info handling

Programming information

struct **sta_info**
STA information

Definition

```
struct sta_info {
  struct list_head list, free_list;
  struct rcu_head rcu_head;
  struct rhlist_head hash_node;
  u8 addr[ETH_ALEN];
  struct ieee80211_local *local;
  struct ieee80211_sub_if_data *sdata;
  struct ieee80211_key __rcu *gtk[NUM_DEFAULT_KEYS +NUM_DEFAULT_MGMT_KEYS␣
↪+ NUM_DEFAULT_BEACON_KEYS];
  struct ieee80211_key __rcu *ptk[NUM_DEFAULT_KEYS];
  u8 ptk_idx;
  struct rate_control_ref *rate_ctrl;
  void *rate_ctrl_priv;
  spinlock_t rate_ctrl_lock;
  spinlock_t lock;
  struct ieee80211_fast_tx __rcu *fast_tx;
  struct ieee80211_fast_rx __rcu *fast_rx;
  struct ieee80211_sta_rx_stats __percpu *pcpu_rx_stats;
#ifdef CONFIG_MAC80211_MESH;
  struct mesh_sta *mesh;
#endif;
  struct work_struct drv_deliver_wk;
  u16 listen_interval;
  bool dead;
  bool removed;
  bool uploaded;
  enum ieee80211_sta_state sta_state;
  unsigned long _flags;
  spinlock_t ps_lock;
  struct sk_buff_head ps_tx_buf[IEEE80211_NUM_ACS];
  struct sk_buff_head tx_filtered[IEEE80211_NUM_ACS];
  unsigned long driver_buffered_tids;
```

(continues on next page)

(continued from previous page)

```
  unsigned long txq_buffered_tids;
  u64 assoc_at;
  long last_connected;
  struct ieee80211_sta_rx_stats rx_stats;
  struct {
    struct ewma_signal signal;
    struct ewma_signal chain_signal[IEEE80211_MAX_CHAINS];
  } rx_stats_avg;
  __le16 last_seq_ctrl[IEEE80211_NUM_TIDS + 1];
  struct {
    unsigned long filtered;
    unsigned long retry_failed, retry_count;
    unsigned int lost_packets;
    unsigned long last_tdls_pkt_time;
    u64 msdu_retries[IEEE80211_NUM_TIDS + 1];
    u64 msdu_failed[IEEE80211_NUM_TIDS + 1];
    unsigned long last_ack;
    s8 last_ack_signal;
    bool ack_signal_filled;
    struct ewma_avg_signal avg_ack_signal;
  } status_stats;
  struct {
    u64 packets[IEEE80211_NUM_ACS];
    u64 bytes[IEEE80211_NUM_ACS];
    struct ieee80211_tx_rate last_rate;
    u64 msdu[IEEE80211_NUM_TIDS + 1];
  } tx_stats;
  u16 tid_seq[IEEE80211_QOS_CTL_TID_MASK + 1];
  struct airtime_info airtime[IEEE80211_NUM_ACS];
  u16 airtime_weight;
  struct sta_ampdu_mlme ampdu_mlme;
#ifdef CONFIG_MAC80211_DEBUGFS;
  struct dentry *debugfs_dir;
#endif;
  enum ieee80211_sta_rx_bandwidth cur_max_bandwidth;
  enum ieee80211_smps_mode known_smps_mode;
  const struct ieee80211_cipher_scheme *cipher_scheme;
  struct codel_params cparams;
  u8 reserved_tid;
  struct cfg80211_chan_def tdls_chandef;
  struct ieee80211_sta sta;
};
```

Members

list global linked list entry

free_list list entry for keeping track of stations to free

rcu_head RCU head used for freeing this station struct

hash_node hash node for rhashtable

addr station's MAC address - duplicated from public part to let the hash table work with just a single cacheline

local pointer to the global information

sdata virtual interface this station belongs to

gtk group keys negotiated with this station, if any

ptk peer keys negotiated with this station, if any

ptk_idx last installed peer key index

rate_ctrl rate control algorithm reference

rate_ctrl_priv rate control private per-STA pointer

rate_ctrl_lock spinlock used to protect rate control data (data inside the algorithm, so serializes calls there)

lock used for locking all fields that require locking, see comments in the header file.

fast_tx TX fastpath information

fast_rx RX fastpath information

pcpu_rx_stats per-CPU RX statistics, assigned only if the driver needs this (by advertising the USES_RSS hw flag)

mesh mesh STA information

drv_deliver_wk used for delivering frames after driver PS unblocking

listen_interval listen interval of this station, when we're acting as AP

dead set to true when sta is unlinked

removed set to true when sta is being removed from sta_list

uploaded set to true when sta is uploaded to the driver

sta_state duplicates information about station state (for debug)

_flags STA flags, see `enum ieee80211_sta_info_flags`, do not use directly

ps_lock used for powersave (when mac80211 is the AP) related locking

ps_tx_buf buffers (per AC) of frames to transmit to this station when it leaves power saving state or polls

tx_filtered buffers (per AC) of frames we already tried to transmit but were filtered by hardware due to STA having entered power saving state, these are also delivered to the station when it leaves powersave or polls for frames

driver_buffered_tids bitmap of TIDs the driver has data buffered on

txq_buffered_tids bitmap of TIDs that mac80211 has txq data buffered on

assoc_at clock boottime (in ns) of last association

last_connected time (in seconds) when a station got connected

rx_stats RX statistics

rx_stats_avg averaged RX statistics

rx_stats_avg.signal averaged signal

rx_stats_avg.chain_signal averaged per-chain signal

last_seq_ctrl last received seq/frag number from this STA (per TID plus one for non-QoS frames)

status_stats TX status statistics

status_stats.filtered # of filtered frames

status_stats.retry_failed # of frames that failed after retry

status_stats.retry_count # of retries attempted

status_stats.lost_packets # of lost packets

status_stats.last_tdls_pkt_time timestamp of last TDLS packet

status_stats.msdu_retries # of MSDU retries

status_stats.msdu_failed # of failed MSDUs

status_stats.last_ack last ack timestamp (jiffies)

status_stats.last_ack_signal last ACK signal

status_stats.ack_signal_filled last ACK signal validity

status_stats.avg_ack_signal average ACK signal

tx_stats TX statistics

tx_stats.packets # of packets transmitted

tx_stats.bytes # of bytes in all packets transmitted

tx_stats.last_rate last TX rate

tx_stats.msdu # of transmitted MSDUs per TID

tid_seq per-TID sequence numbers for sending to this STA

airtime per-AC struct airtime_info describing airtime statistics for this station

airtime_weight station weight for airtime fairness calculation purposes

ampdu_mlme A-MPDU state machine state

debugfs_dir debug filesystem directory dentry

cur_max_bandwidth maximum bandwidth to use for TX to the station, taken from HT/VHT capabilities or VHT operating mode notification

known_smps_mode the smps_mode the client thinks we are in. Relevant for AP only.

cipher_scheme optional cipher scheme for this station

cparams CoDel parameters for this station.

reserved_tid reserved TID (if any, otherwise IEEE80211_TID_UNRESERVED)

tdls_chandef a TDLS peer can have a wider chandef that is compatible to the BSS one.

sta station information we share with the driver

Description

This structure collects information about a station that mac80211 is communicating with.

enum **ieee80211_sta_info_flags**
Stations flags

Constants

WLAN_STA_AUTH Station is authenticated.

WLAN_STA_ASSOC Station is associated.

WLAN_STA_PS_STA Station is in power-save mode

WLAN_STA_AUTHORIZED Station is authorized to send/receive traffic. This bit is always checked so needs to be enabled for all stations when virtual port control is not in use.

WLAN_STA_SHORT_PREAMBLE Station is capable of receiving short-preamble frames.

WLAN_STA_WDS Station is one of our WDS peers.

WLAN_STA_CLEAR_PS_FILT Clear PS filter in hardware (using the IEEE80211_TX_CTL_CLEAR_PS_FILT control flag) when the next frame to this station is transmitted.

WLAN_STA_MFP Management frame protection is used with this STA.

WLAN_STA_BLOCK_BA Used to deny ADDBA requests (both TX and RX) during suspend/resume and station removal.

WLAN_STA_PS_DRIVER driver requires keeping this station in power-save mode logically to flush frames that might still be in the queues

WLAN_STA_PSPOLL Station sent PS-poll while driver was keeping station in power-save mode, reply when the driver unblocks.

WLAN_STA_TDLS_PEER Station is a TDLS peer.

WLAN_STA_TDLS_PEER_AUTH This TDLS peer is authorized to send direct packets. This means the link is enabled.

WLAN_STA_TDLS_INITIATOR We are the initiator of the TDLS link with this station.

WLAN_STA_TDLS_CHAN_SWITCH This TDLS peer supports TDLS channel-switching

WLAN_STA_TDLS_OFF_CHANNEL The local STA is currently off-channel with this TDLS peer

WLAN_STA_TDLS_WIDER_BW This TDLS peer supports working on a wider bw on the BSS base channel.

WLAN_STA_UAPSD Station requested unscheduled SP while driver was keeping station in power-save mode, reply when the driver unblocks the station.

WLAN_STA_SP Station is in a service period, so don’ t try to reply to other uAPSD trigger frames or PS-Poll.

WLAN_STA_4ADDR_EVENT 4-addr event was already sent for this frame.

WLAN_STA_INSERTED This station is inserted into the hash table.

WLAN_STA_RATE_CONTROL rate control was initialized for this station.

WLAN_STA_TOFFSET_KNOWN toffset calculated for this station is valid.

WLAN_STA_MPSP_OWNER local STA is owner of a mesh Peer Service Period.

WLAN_STA_MPSP_RECIPIENT local STA is recipient of a MPSP.

WLAN_STA_PS_DELIVER station woke up, but we’re still blocking TX until pending frames are delivered

WLAN_STA_USES_ENCRYPTION This station was configured for encryption, so drop all packets without a key later.

NUM_WLAN_STA_FLAGS number of defined flags

Description

These flags are used with `struct sta_info`’s **flags** member, but only indirectly with set_sta_flag() and friends.

STA information lifetime rules

STA info structures (`struct sta_info`) are managed in a hash table for faster lookup and a list for iteration. They are managed using RCU, i.e. access to the list and hash table is protected by RCU.

Upon allocating a STA info structure with sta_info_alloc(), the caller owns that structure. It must then insert it into the hash table using either sta_info_insert() or sta_info_insert_rcu(); only in the latter case (which acquires an rcu read section but must not be called from within one) will the pointer still be valid after the call. Note that the caller may not do much with the STA info before inserting it, in particular, it may not start any mesh peer link management or add encryption keys.

When the insertion fails (sta_info_insert()) returns non-zero), the structure will have been freed by sta_info_insert()!

Station entries are added by mac80211 when you establish a link with a peer. This means different things for the different type of interfaces we support. For a regular station this mean we add the AP sta when we receive an association response from the AP. For IBSS this occurs when get to know about a peer on the same IBSS. For WDS we add the sta for the peer immediately upon device open. When using AP mode we add stations for each respective station upon request from userspace through nl80211.

In order to remove a STA info structure, various sta_info_destroy_*() calls are available.

There is no concept of ownership on a STA entry, each structure is owned by the global hash table/list until it is removed. All users of the structure need to be RCU protected so that the structure won’t be freed before they are done using it.

47.4.17 Aggregation Functions

struct **sta_ampdu_mlme**
STA aggregation information.

Definition

```
struct sta_ampdu_mlme {
  struct mutex mtx;
  struct tid_ampdu_rx __rcu *tid_rx[IEEE80211_NUM_TIDS];
  u8 tid_rx_token[IEEE80211_NUM_TIDS];
  unsigned long tid_rx_timer_expired[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  unsigned long tid_rx_stop_requested[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  unsigned long tid_rx_manage_offl[BITS_TO_LONGS(2 * IEEE80211_NUM_TIDS)];
  unsigned long agg_session_valid[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  unsigned long unexpected_agg[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  struct work_struct work;
  struct tid_ampdu_tx __rcu *tid_tx[IEEE80211_NUM_TIDS];
  struct tid_ampdu_tx *tid_start_tx[IEEE80211_NUM_TIDS];
  unsigned long last_addba_req_time[IEEE80211_NUM_TIDS];
  u8 addba_req_num[IEEE80211_NUM_TIDS];
  u8 dialog_token_allocator;
};
```

Members

mtx mutex to protect all TX data (except non-NULL assignments to tid_tx[idx], which are protected by the sta spinlock) tid_start_tx is also protected by sta->lock.

tid_rx aggregation info for Rx per TID – RCU protected

tid_rx_token dialog tokens for valid aggregation sessions

tid_rx_timer_expired bitmap indicating on which TIDs the RX timer expired until the work for it runs

tid_rx_stop_requested bitmap indicating which BA sessions per TID the driver requested to close until the work for it runs

tid_rx_manage_offl bitmap indicating which BA sessions were requested to be treated as started/stopped due to offloading

agg_session_valid bitmap indicating which TID has a rx BA session open on

unexpected_agg bitmap indicating which TID already sent a delBA due to unexpected aggregation related frames outside a session

work work struct for starting/stopping aggregation

tid_tx aggregation info for Tx per TID

tid_start_tx sessions where start was requested

last_addba_req_time timestamp of the last addBA request.

addba_req_num number of times addBA request has been sent.

dialog_token_allocator dialog token enumerator for each new session;

struct **tid_ampdu_tx**
TID aggregation information (Tx).

Definition

```
struct tid_ampdu_tx {
  struct rcu_head rcu_head;
  struct timer_list session_timer;
  struct timer_list addba_resp_timer;
  struct sk_buff_head pending;
  struct sta_info *sta;
  unsigned long state;
  unsigned long last_tx;
  u16 timeout;
  u8 dialog_token;
  u8 stop_initiator;
  bool tx_stop;
  u16 buf_size;
  u16 failed_bar_ssn;
  bool bar_pending;
  bool amsdu;
  u8 tid;
};
```

Members

rcu_head rcu head for freeing structure

session_timer check if we keep Tx-ing on the TID (by timeout value)

addba_resp_timer timer for peer's response to addba request

pending pending frames queue – use sta's spinlock to protect

sta station we are attached to

state session state (see above)

last_tx jiffies of last tx activity

timeout session timeout value to be filled in ADDBA requests

dialog_token dialog token for aggregation session

stop_initiator initiator of a session stop

tx_stop TX DelBA frame when stopping

buf_size reorder buffer size at receiver

failed_bar_ssn ssn of the last failed BAR tx attempt

bar_pending BAR needs to be re-sent

amsdu support A-MSDU withing A-MDPU

tid TID number

Description

This structure's lifetime is managed by RCU, assignments to the array holding it must hold the aggregation mutex.

The TX path can access it under RCU lock-free if, and only if, the state has the flag HT_AGG_STATE_OPERATIONAL set. Otherwise, the TX path must also acquire the spinlock and re-check the state, see comments in the tx code touching it.

struct **tid_ampdu_rx**
TID aggregation information (Rx).

Definition

```
struct tid_ampdu_rx {
  struct rcu_head rcu_head;
  spinlock_t reorder_lock;
  u64 reorder_buf_filtered;
  struct sk_buff_head *reorder_buf;
  unsigned long *reorder_time;
  struct sta_info *sta;
  struct timer_list session_timer;
  struct timer_list reorder_timer;
  unsigned long last_rx;
  u16 head_seq_num;
  u16 stored_mpdu_num;
  u16 ssn;
  u16 buf_size;
  u16 timeout;
  u8 tid;
  u8 auto_seq:1,removed:1, started:1;
};
```

Members

rcu_head RCU head used for freeing this struct

reorder_lock serializes access to reorder buffer, see below.

reorder_buf_filtered bitmap indicating where there are filtered frames in the reorder buffer that should be ignored when releasing frames

reorder_buf buffer to reorder incoming aggregated MPDUs. An MPDU may be an A-MSDU with individually reported subframes.

reorder_time jiffies when skb was added

sta station we are attached to

session_timer check if peer keeps Tx-ing on the TID (by timeout value)

reorder_timer releases expired frames from the reorder buffer.

last_rx jiffies of last rx activity

head_seq_num head sequence number in reordering buffer.

stored_mpdu_num number of MPDUs in reordering buffer

ssn Starting Sequence Number expected to be aggregated.

buf_size buffer size for incoming A-MPDUs

timeout reset timer value (in TUs).

tid TID number

auto_seq used for offloaded BA sessions to automatically pick head_seq_and and ssn.

removed this session is removed (but might have been found due to RCU)

started this session has started (head ssn or higher was received)

Description

This structure's lifetime is managed by RCU, assignments to the array holding it must hold the aggregation mutex.

The **reorder_lock** is used to protect the members of this struct, except for **timeout**, **buf_size** and **dialog_token**, which are constant across the lifetime of the struct (the dialog token being used only for debugging).

47.4.18 Synchronisation Functions

TBD

Locking, lots of RCU

CHAPTER

FORTYEIGHT

THE USERSPACE I/O HOWTO

Author Hans-Jürgen Koch Linux developer, Linutronix

Date 2006-12-11

48.1 About this document

48.1.1 Translations

If you know of any translations for this document, or you are interested in translating it, please email me hjk@hansjkoch.de.

48.1.2 Preface

For many types of devices, creating a Linux kernel driver is overkill. All that is really needed is some way to handle an interrupt and provide access to the memory space of the device. The logic of controlling the device does not necessarily have to be within the kernel, as the device does not need to take advantage of any of other resources that the kernel provides. One such common class of devices that are like this are for industrial I/O cards.

To address this situation, the userspace I/O system (UIO) was designed. For typical industrial I/O cards, only a very small kernel module is needed. The main part of the driver will run in user space. This simplifies development and reduces the risk of serious bugs within a kernel module.

Please note that UIO is not an universal driver interface. Devices that are already handled well by other kernel subsystems (like networking or serial or USB) are no candidates for an UIO driver. Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped. The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

48.1.3 Acknowledgments

I'd like to thank Thomas Gleixner and Benedikt Spranger of Linutronix, who have not only written most of the UIO code, but also helped greatly writing this HOWTO by giving me all kinds of background information.

48.1.4 Feedback

Find something wrong with this document? (Or perhaps something right?) I would love to hear from you. Please email me at hjk@hansjkoch.de.

48.2 About UIO

If you use UIO for your card's driver, here's what you get:

- only one small kernel module to write and maintain.
- develop the main part of your driver in user space, with all the tools and libraries you're used to.
- bugs in your driver won't crash the kernel.
- updates of your driver can take place without recompiling the kernel.

48.2.1 How UIO works

Each UIO device is accessed through a device file and several sysfs attribute files. The device file will be called `/dev/uio0` for the first device, and `/dev/uio1`, `/dev/uio2` and so on for subsequent devices.

`/dev/uioX` is used to access the address space of the card. Just use `mmap()` to access registers or RAM locations of your card.

Interrupts are handled by reading from `/dev/uioX`. A blocking `read()` from `/dev/uioX` will return as soon as an interrupt occurs. You can also use `select()` on `/dev/uioX` to wait for an interrupt. The integer value read from `/dev/uioX` represents the total interrupt count. You can use this number to figure out if you missed some interrupts.

For some hardware that has more than one interrupt source internally, but not separate IRQ mask and status registers, there might be situations where userspace cannot determine what the interrupt source was if the kernel handler disables them by writing to the chip's IRQ register. In such a case, the kernel has to disable the IRQ completely to leave the chip's register untouched. Now the userspace part can determine the cause of the interrupt, but it cannot re-enable interrupts. Another cornercase is chips where re-enabling interrupts is a read-modify-write operation to a combined IRQ status/acknowledge register. This would be racy if a new interrupt occurred simultaneously.

To address these problems, UIO also implements a write() function. It is normally not used and can be ignored for hardware that has only a single interrupt source

or has separate IRQ mask and status registers. If you need it, however, a write to /dev/uioX will call the irqcontrol() function implemented by the driver. You have to write a 32-bit value that is usually either 0 or 1 to disable or enable interrupts. If a driver does not implement irqcontrol(), write() will return with -ENOSYS.

To handle interrupts properly, your custom kernel module can provide its own interrupt handler. It will automatically be called by the built-in handler.

For cards that don't generate interrupts but need to be polled, there is the possibility to set up a timer that triggers the interrupt handler at configurable time intervals. This interrupt simulation is done by calling uio_event_notify() from the timer's event handler.

Each driver provides attributes that are used to read or write variables. These attributes are accessible through sysfs files. A custom kernel driver module can add its own attributes to the device owned by the uio driver, but not added to the UIO device itself at this time. This might change in the future if it would be found to be useful.

The following standard attributes are provided by the UIO framework:

- name: The name of your device. It is recommended to use the name of your kernel module for this.
- version: A version string defined by your driver. This allows the user space part of your driver to deal with different versions of the kernel module.
- event: The total number of interrupts handled by the driver since the last time the device node was read.

These attributes appear under the /sys/class/uio/uioX directory. Please note that this directory might be a symlink, and not a real directory. Any userspace code that accesses it must be able to handle this.

Each UIO device can make one or more memory regions available for memory mapping. This is necessary because some industrial I/O cards require access to more than one PCI memory region in a driver.

Each mapping has its own directory in sysfs, the first mapping appears as /sys/class/uio/uioX/maps/map0/. Subsequent mappings create directories map1/, map2/, and so on. These directories will only appear if the size of the mapping is not 0.

Each mapX/ directory contains four read-only files that show attributes of the memory:

- name: A string identifier for this mapping. This is optional, the string can be empty. Drivers can set this to make it easier for userspace to find the correct mapping.
- addr: The address of memory that can be mapped.
- size: The size, in bytes, of the memory pointed to by addr.
- offset: The offset, in bytes, that has to be added to the pointer returned by mmap() to get to the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by mmap() are always page aligned, so it is good style to always add this offset.

From userspace, the different mappings are distinguished by adjusting the `offset` parameter of the `mmap()` call. To map the memory of mapping N, you have to use N times the page size as your offset:

```
offset = N * getpagesize();
```

Sometimes there is hardware with memory-like regions that can not be mapped with the technique described here, but there are still ways to access them from userspace. The most common example are x86 ioports. On x86 systems, userspace can access these ioports using `ioperm()`, `iopl()`, `inb()`, `outb()`, and similar functions.

Since these ioport regions can not be mapped, they will not appear under `/sys/class/uio/uioX/maps/` like the normal memory described above. Without information about the port regions a hardware has to offer, it becomes difficult for the userspace part of the driver to find out which ports belong to which UIO device.

To address this situation, the new directory `/sys/class/uio/uioX/portio/` was added. It only exists if the driver wants to pass information about one or more port regions to userspace. If that is the case, subdirectories named `port0`, `port1`, and so on, will appear underneath `/sys/class/uio/uioX/portio/`.

Each `portX/` directory contains four read-only files that show name, start, size, and type of the port region:

- `name`: A string identifier for this port region. The string is optional and can be empty. Drivers can set it to make it easier for userspace to find a certain port region.
- `start`: The first port of this region.
- `size`: The number of ports in this region.
- `porttype`: A string describing the type of port.

48.3 Writing your own kernel module

Please have a look at `uio_cif.c` as an example. The following paragraphs explain the different sections of this file.

48.3.1 struct uio_info

This structure tells the framework the details of your driver, Some of the members are required, others are optional.

- `const char *name`: Required. The name of your driver as it will appear in sysfs. I recommend using the name of your module for this.
- `const char *version`: Required. This string appears in `/sys/class/uio/uioX/version`.
- `struct uio_mem mem[ MAX_UIO_MAPS ]`: Required if you have memory that can be mapped with `mmap()`. For each mapping you need to fill one of the `uio_mem` structures. See the description below for details.

- `struct uio_port port[ MAX_UIO_PORTS_REGIONS ]`: Required if you want to pass information about ioports to userspace. For each port region you need to fill one of the `uio_port` structures. See the description below for details.
- `long irq`: Required. If your hardware generates an interrupt, it's your modules task to determine the irq number during initialization. If you don't have a hardware generated interrupt but want to trigger the interrupt handler in some other way, set `irq` to `UIO_IRQ_CUSTOM`. If you had no interrupt at all, you could set `irq` to `UIO_IRQ_NONE`, though this rarely makes sense.
- `unsigned long irq_flags`: Required if you've set `irq` to a hardware interrupt number. The flags given here will be used in the call to `request_irq()`.
- `int (*mmap)(struct uio_info *info, struct vm_area_struct *vma)`: Optional. If you need a special `mmap()` function, you can set it here. If this pointer is not NULL, your `mmap()` will be called instead of the built-in one.
- `int (*open)(struct uio_info *info, struct inode *inode)`: Optional. You might want to have your own `open()`, e.g. to enable interrupts only when your device is actually used.
- `int (*release)(struct uio_info *info, struct inode *inode)`: Optional. If you define your own `open()`, you will probably also want a custom `release()` function.
- `int (*irqcontrol)(struct uio_info *info, s32 irq_on)`: Optional. If you need to be able to enable or disable interrupts from userspace by writing to `/dev/uioX`, you can implement this function. The parameter `irq_on` will be 0 to disable interrupts and 1 to enable them.

Usually, your device will have one or more memory regions that can be mapped to user space. For each region, you have to set up a `struct uio_mem` in the `mem[]` array. Here's a description of the fields of `struct uio_mem`:

- `const char *name`: Optional. Set this to help identify the memory region, it will show up in the corresponding sysfs node.
- `int memtype`: Required if the mapping is used. Set this to `UIO_MEM_PHYS` if you you have physical memory on your card to be mapped. Use `UIO_MEM_LOGICAL` for logical memory (e.g. allocated with `__get_free_pages()` but not kmalloc()). There's also `UIO_MEM_VIRTUAL` for virtual memory.
- `phys_addr_t addr`: Required if the mapping is used. Fill in the address of your memory block. This address is the one that appears in sysfs.
- `resource_size_t size`: Fill in the size of the memory block that `addr` points to. If `size` is zero, the mapping is considered unused. Note that you must initialize `size` with zero for all unused mappings.
- `void *internal_addr`: If you have to access this memory region from within your kernel module, you will want to map it internally by using something like `ioremap()`. Addresses returned by this function cannot be mapped to user space, so you must not store it in `addr`. Use `internal_addr` instead to remember such an address.

Please do not touch the `map` element of `struct uio_mem`! It is used by the UIO framework to set up sysfs files for this mapping. Simply leave it alone.

Sometimes, your device can have one or more port regions which can not be mapped to userspace. But if there are other possibilities for userspace to access these ports, it makes sense to make information about the ports available in sysfs. For each region, you have to set up a `struct uio_port` in the `port[]` array. Here's a description of the fields of `struct uio_port`:

- `char *porttype`: Required. Set this to one of the predefined constants. Use `UIO_PORT_X86` for the ioports found in x86 architectures.
- `unsigned long start`: Required if the port region is used. Fill in the number of the first port of this region.
- `unsigned long size`: Fill in the number of ports in this region. If `size` is zero, the region is considered unused. Note that you must initialize `size` with zero for all unused regions.

Please do not touch the `portio` element of `struct uio_port`! It is used internally by the UIO framework to set up sysfs files for this region. Simply leave it alone.

48.3.2 Adding an interrupt handler

What you need to do in your interrupt handler depends on your hardware and on how you want to handle it. You should try to keep the amount of code in your kernel interrupt handler low. If your hardware requires no action that you have to perform after each interrupt, then your handler can be empty.

If, on the other hand, your hardware needs some action to be performed after each interrupt, then you must do it in your kernel module. Note that you cannot rely on the userspace part of your driver. Your userspace program can terminate at any time, possibly leaving your hardware in a state where proper interrupt handling is still required.

There might also be applications where you want to read data from your hardware at each interrupt and buffer it in a piece of kernel memory you've allocated for that purpose. With this technique you could avoid loss of data if your userspace program misses an interrupt.

A note on shared interrupts: Your driver should support interrupt sharing whenever this is possible. It is possible if and only if your driver can detect whether your hardware has triggered the interrupt or not. This is usually done by looking at an interrupt status register. If your driver sees that the IRQ bit is actually set, it will perform its actions, and the handler returns IRQ_HANDLED. If the driver detects that it was not your hardware that caused the interrupt, it will do nothing and return IRQ_NONE, allowing the kernel to call the next possible interrupt handler.

If you decide not to support shared interrupts, your card won't work in computers with no free interrupts. As this frequently happens on the PC platform, you can save yourself a lot of trouble by supporting interrupt sharing.

48.3.3 Using uio_pdrv for platform devices

In many cases, UIO drivers for platform devices can be handled in a generic way. In the same place where you define your `struct platform_device`, you simply also implement your interrupt handler and fill your `struct uio_info`. A pointer to this `struct uio_info` is then used as `platform_data` for your platform device.

You also need to set up an array of `struct resource` containing addresses and sizes of your memory mappings. This information is passed to the driver using the `.resource` and `.num_resources` elements of `struct platform_device`.

You now have to set the `.name` element of `struct platform_device` to `"uio_pdrv"` to use the generic UIO platform device driver. This driver will fill the `mem[]` array according to the resources given, and register the device.

The advantage of this approach is that you only have to edit a file you need to edit anyway. You do not have to create an extra driver.

48.3.4 Using uio_pdrv_genirq for platform devices

Especially in embedded devices, you frequently find chips where the irq pin is tied to its own dedicated interrupt line. In such cases, where you can be really sure the interrupt is not shared, we can take the concept of `uio_pdrv` one step further and use a generic interrupt handler. That's what `uio_pdrv_genirq` does.

The setup for this driver is the same as described above for `uio_pdrv`, except that you do not implement an interrupt handler. The `.handler` element of `struct uio_info` must remain `NULL`. The `.irq_flags` element must not contain `IRQF_SHARED`.

You will set the `.name` element of `struct platform_device` to `"uio_pdrv_genirq"` to use this driver.

The generic interrupt handler of `uio_pdrv_genirq` will simply disable the interrupt line using `disable_irq_nosync()`. After doing its work, userspace can reenable the interrupt by writing 0x00000001 to the UIO device file. The driver already implements an `irq_control()` to make this possible, you must not implement your own.

Using `uio_pdrv_genirq` not only saves a few lines of interrupt handler code. You also do not need to know anything about the chip's internal registers to create the kernel part of the driver. All you need to know is the irq number of the pin the chip is connected to.

When used in a device-tree enabled system, the driver needs to be probed with the `"of_id"` module parameter set to the `"compatible"` string of the node the driver is supposed to handle. By default, the node's name (without the unit address) is exposed as name for the UIO device in userspace. To set a custom name, a property named `"linux,uio-name"` may be specified in the DT node.

48.3.5 Using uio_dmem_genirq for platform devices

In addition to statically allocated memory ranges, they may also be a desire to use dynamically allocated regions in a user space driver. In particular, being able to access memory made available through the dma-mapping API, may be particularly useful. The `uio_dmem_genirq` driver provides a way to accomplish this.

This driver is used in a similar manner to the `"uio_pdrv_genirq"` driver with respect to interrupt configuration and handling.

Set the `.name` element of `struct platform_device` to `"uio_dmem_genirq"` to use this driver.

When using this driver, fill in the `.platform_data` element of `struct platform_device`, which is of type `struct uio_dmem_genirq_pdata` and which contains the following elements:

- `struct uio_info uioinfo`: The same structure used as the `uio_pdrv_genirq` platform data
- `unsigned int *dynamic_region_sizes`: Pointer to list of sizes of dynamic memory regions to be mapped into user space.
- `unsigned int num_dynamic_regions`: Number of elements in `dynamic_region_sizes` array.

The dynamic regions defined in the platform data will be appended to the `` mem[] `` array after the platform device resources, which implies that the total number of static and dynamic memory regions cannot exceed `MAX_UIO_MAPS`.

The dynamic memory regions will be allocated when the UIO device file, `/dev/uioX` is opened. Similar to static memory resources, the memory region information for dynamic regions is then visible via sysfs at `/sys/class/uio/uioX/maps/mapY/*`. The dynamic memory regions will be freed when the UIO device file is closed. When no processes are holding the device file open, the address returned to userspace is ~0.

48.4 Writing a driver in userspace

Once you have a working kernel module for your hardware, you can write the userspace part of your driver. You don’t need any special libraries, your driver can be written in any reasonable language, you can use floating point numbers and so on. In short, you can use all the tools and libraries you’d normally use for writing a userspace application.

48.4.1 Getting information about your UIO device

Information about all UIO devices is available in sysfs. The first thing you should do in your driver is check `name` and `version` to make sure you're talking to the right device and that its kernel driver has the version you expect.

You should also make sure that the memory mapping you need exists and has the size you expect.

There is a tool called `lsuio` that lists UIO devices and their attributes. It is available here:

http://www.osadl.org/projects/downloads/UIO/user/

With `lsuio` you can quickly check if your kernel module is loaded and which attributes it exports. Have a look at the manpage for details.

The source code of `lsuio` can serve as an example for getting information about an UIO device. The file `uio_helper.c` contains a lot of functions you could use in your userspace driver code.

48.4.2 mmap() device memory

After you made sure you've got the right device with the memory mappings you need, all you have to do is to call `mmap()` to map the device's memory to userspace.

The parameter `offset` of the `mmap()` call has a special meaning for UIO devices: It is used to select which mapping of your device you want to map. To map the memory of mapping N, you have to use N times the page size as your offset:

```
offset = N * getpagesize();
```

N starts from zero, so if you've got only one memory range to map, set `offset = 0`. A drawback of this technique is that memory is always mapped beginning with its start address.

48.4.3 Waiting for interrupts

After you successfully mapped your devices memory, you can access it like an ordinary array. Usually, you will perform some initialization. After that, your hardware starts working and will generate an interrupt as soon as it's finished, has some data available, or needs your attention because an error occurred.

`/dev/uioX` is a read-only file. A `read()` will always block until an interrupt occurs. There is only one legal value for the `count` parameter of `read()`, and that is the size of a signed 32 bit integer (4). Any other value for `count` causes `read()` to fail. The signed 32 bit integer read is the interrupt count of your device. If the value is one more than the value you read the last time, everything is OK. If the difference is greater than one, you missed interrupts.

You can also use `select()` on `/dev/uioX`.

48.5 Generic PCI UIO driver

The generic driver is a kernel module named uio_pci_generic. It can work with any device compliant to PCI 2.3 (circa 2002) and any compliant PCI Express device. Using this, you only need to write the userspace driver, removing the need to write a hardware-specific kernel module.

48.5.1 Making the driver recognize the device

Since the driver does not declare any device ids, it will not get loaded automatically and will not automatically bind to any devices, you must load it and allocate id to the driver yourself. For example:

```
modprobe uio_pci_generic
echo "8086 10f5" > /sys/bus/pci/drivers/uio_pci_generic/new_id
```

If there already is a hardware specific kernel driver for your device, the generic driver still won't bind to it, in this case if you want to use the generic driver (why would you?) you'll have to manually unbind the hardware specific driver and bind the generic driver, like this:

```
echo -n 0000:00:19.0 > /sys/bus/pci/drivers/e1000e/unbind
echo -n 0000:00:19.0 > /sys/bus/pci/drivers/uio_pci_generic/bind
```

You can verify that the device has been bound to the driver by looking for it in sysfs, for example like the following:

```
ls -l /sys/bus/pci/devices/0000:00:19.0/driver
```

Which if successful should print:

```
.../0000:00:19.0/driver -> ../../../bus/pci/drivers/uio_pci_generic
```

Note that the generic driver will not bind to old PCI 2.2 devices. If binding the device failed, run the following command:

```
dmesg
```

and look in the output for failure reasons.

48.5.2 Things to know about uio_pci_generic

Interrupts are handled using the Interrupt Disable bit in the PCI command register and Interrupt Status bit in the PCI status register. All devices compliant to PCI 2.3 (circa 2002) and all compliant PCI Express devices should support these bits. uio_pci_generic detects this support, and won't bind to devices which do not support the Interrupt Disable Bit in the command register.

On each interrupt, uio_pci_generic sets the Interrupt Disable bit. This prevents the device from generating further interrupts until the bit is cleared. The userspace driver should clear this bit before blocking and waiting for more interrupts.

48.5.3 Writing userspace driver using uio_pci_generic

Userspace driver can use pci sysfs interface, or the libpci library that wraps it, to talk to the device and to re-enable interrupts by writing to the command register.

48.5.4 Example code using uio_pci_generic

Here is some sample userspace driver code using uio_pci_generic:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main()
{
    int uiofd;
    int configfd;
    int err;
    int i;
    unsigned icount;
    unsigned char command_high;

    uiofd = open("/dev/uio0", O_RDONLY);
    if (uiofd < 0) {
        perror("uio open:");
        return errno;
    }
    configfd = open("/sys/class/uio/uio0/device/config", O_RDWR);
    if (configfd < 0) {
        perror("config open:");
        return errno;
    }

    /* Read and cache command value */
    err = pread(configfd, &command_high, 1, 5);
    if (err != 1) {
        perror("command config read:");
        return errno;
    }
    command_high &= ~0x4;

    for(i = 0;; ++i) {
        /* Print out a message, for debugging. */
        if (i == 0)
            fprintf(stderr, "Started uio test driver.\n");
        else
            fprintf(stderr, "Interrupts: %d\n", icount);

        /****************************************/
        /* Here we got an interrupt from the
           device. Do something to it. */
```

(continues on next page)

(continued from previous page)

```
        /*************************************/

        /* Re-enable interrupts. */
        err = pwrite(configfd, &command_high, 1, 5);
        if (err != 1) {
            perror("config write:");
            break;
        }

        /* Wait for next interrupt. */
        err = read(uiofd, &icount, 4);
        if (err != 4) {
            perror("uio read:");
            break;
        }

    }
    return errno;
}
```

48.6 Generic Hyper-V UIO driver

The generic driver is a kernel module named uio_hv_generic. It supports devices on the Hyper-V VMBus similar to uio_pci_generic on PCI bus.

48.6.1 Making the driver recognize the device

Since the driver does not declare any device GUID's, it will not get loaded automatically and will not automatically bind to any devices, you must load it and allocate id to the driver yourself. For example, to use the network device class GUID:

```
modprobe uio_hv_generic
echo "f8615163-df3e-46c5-913f-f2d2f965ed0e" > /sys/bus/vmbus/drivers/uio_
↪hv_generic/new_id
```

If there already is a hardware specific kernel driver for the device, the generic driver still won't bind to it, in this case if you want to use the generic driver for a userspace library you'll have to manually unbind the hardware specific driver and bind the generic driver, using the device specific GUID like this:

```
echo -n ed963694-e847-4b2a-85af-bc9cfc11d6f3 > /sys/bus/vmbus/drivers/hv_
↪netvsc/unbind
echo -n ed963694-e847-4b2a-85af-bc9cfc11d6f3 > /sys/bus/vmbus/drivers/uio_
↪hv_generic/bind
```

You can verify that the device has been bound to the driver by looking for it in sysfs, for example like the following:

```
ls -l /sys/bus/vmbus/devices/ed963694-e847-4b2a-85af-bc9cfc11d6f3/driver
```

Which if successful should print:

```
.../ed963694-e847-4b2a-85af-bc9cfc11d6f3/driver -> ../../../bus/vmbus/
↪drivers/uio_hv_generic
```

48.6.2 Things to know about uio_hv_generic

On each interrupt, uio_hv_generic sets the Interrupt Disable bit. This prevents the device from generating further interrupts until the bit is cleared. The userspace driver should clear this bit before blocking and waiting for more interrupts.

When host rescinds a device, the interrupt file descriptor is marked down and any reads of the interrupt file descriptor will return -EIO. Similar to a closed socket or disconnected serial device.

The vmbus device regions are mapped into uio device resources:

0) Channel ring buffers: guest to host and host to guest

1) Guest to host interrupt signalling pages

2) Guest to host monitor page

3) Network receive buffer region

4) Network send buffer region

If a subchannel is created by a request to host, then the uio_hv_generic device driver will create a sysfs binary file for the per-channel ring buffer. For example:

```
/sys/bus/vmbus/devices/3811fe4d-0fa0-4b62-981a-74fc1084c757/channels/21/
↪ring
```

48.7 Further information

- OSADL homepage.
- Linutronix homepage.

CHAPTER

FORTYNINE

LINUX FIRMWARE API

49.1 Introduction

The firmware API enables kernel code to request files required for functionality from userspace, the uses vary:

- Microcode for CPU errata
- Device driver firmware, required to be loaded onto device microcontrollers
- Device driver information data (calibration data, EEPROM overrides), some of which can be completely optional.

49.1.1 Types of firmware requests

There are two types of calls:

- Synchronous
- Asynchronous

Which one you use vary depending on your requirements, the rule of thumb however is you should strive to use the asynchronous APIs unless you also are already using asynchronous initialization mechanisms which will not stall or delay boot. Even if loading firmware does not take a lot of time processing firmware might, and this can still delay boot or initialization, as such mechanisms such as asynchronous probe can help supplement drivers.

49.2 Firmware API core features

The firmware API has a rich set of core features available. This section documents these features.

49.2.1 Firmware search paths

The following search paths are used to look for firmware on your root filesystem.

- fw_path_para - module parameter - default is empty so this is ignored
- /lib/firmware/updates/UTS_RELEASE/
- /lib/firmware/updates/
- /lib/firmware/UTS_RELEASE/
- /lib/firmware/

The module parameter “path” can be passed to the firmware_class module to activate the first optional custom fw_path_para. The custom path can only be up to 256 characters long. The kernel parameter passed would be:

- ‘firmware_class.path=$CUSTOMIZED_PATH’

There is an alternative to customize the path at run time after bootup, you can use the file:

- /sys/module/firmware_class/parameters/path

You would echo into it your custom path and firmware requested will be searched for there first.

49.2.2 Built-in firmware

Firmware can be built-in to the kernel, this means building the firmware into vmlinux directly, to enable avoiding having to look for firmware from the filesystem. Instead, firmware can be looked for inside the kernel directly. You can enable built-in firmware using the kernel configuration options:

- CONFIG_EXTRA_FIRMWARE
- CONFIG_EXTRA_FIRMWARE_DIR

There are a few reasons why you might want to consider building your firmware into the kernel with CONFIG_EXTRA_FIRMWARE:

- Speed
- Firmware is needed for accessing the boot device, and the user doesn’t want to stuff the firmware into the boot initramfs.

Even if you have these needs there are a few reasons why you may not be able to make use of built-in firmware:

- Legalese - firmware is non-GPL compatible
- Some firmware may be optional
- Firmware upgrades are possible, therefore a new firmware would implicate a complete kernel rebuild.
- Some firmware files may be really large in size. The remote-proc subsystem is an example subsystem which deals with these sorts of firmware

- The firmware may need to be scraped out from some device specific location dynamically, an example is calibration data for for some WiFi chipsets. This calibration data can be unique per sold device.

49.2.3 Firmware cache

When Linux resumes from suspend some device drivers require firmware lookups to re-initialize devices. During resume there may be a period of time during which firmware lookups are not possible, during this short period of time firmware requests will fail. Time is of essence though, and delaying drivers to wait for the root filesystem for firmware delays user experience with device functionality. In order to support these requirements the firmware infrastructure implements a firmware cache for device drivers for most API calls, automatically behind the scenes.

The firmware cache makes using certain firmware API calls safe during a device driver' s suspend and resume callback. Users of these API calls needn' t cache the firmware by themselves for dealing with firmware loss during system resume.

The firmware cache works by requesting for firmware prior to suspend and caching it in memory. Upon resume device drivers using the firmware API will have access to the firmware immediately, without having to wait for the root filesystem to mount or dealing with possible race issues with lookups as the root filesystem mounts.

Some implementation details about the firmware cache setup:

- The firmware cache is setup by adding a devres entry for each device that uses all synchronous call except `request_firmware_into_buf()`.
- If an asynchronous call is used the firmware cache is only set up for a device if if the second argument (uevent) to `request_firmware_nowait()` is true. When uevent is true it requests that a kobject uevent be sent to userspace for the firmware request through the sysfs fallback mechanism if the firmware file is not found.
- If the firmware cache is determined to be needed as per the above two criteria the firmware cache is setup by adding a devres entry for the device making the firmware request.
- The firmware devres entry is maintained throughout the lifetime of the device. This means that even if you `release_firmware()` the firmware cache will still be used on resume from suspend.
- The timeout for the fallback mechanism is temporarily reduced to 10 seconds as the firmware cache is set up during suspend, the timeout is set back to the old value you had configured after the cache is set up.
- Upon suspend any pending non-uevent firmware requests are killed to avoid stalling the kernel, this is done with kill_requests_without_uevent(). Kernel calls requiring the non-uevent therefore need to implement their own firmware cache mechanism but must not use the firmware API on suspend.

49.2.4 Direct filesystem lookup

Direct filesystem lookup is the most common form of firmware lookup performed by the kernel. The kernel looks for the firmware directly on the root filesystem in the paths documented in the section 'Firmware search paths'. The filesystem lookup is implemented in fw_get_filesystem_firmware(), it uses common core kernel file loader facility kernel_read_file_from_path(). The max path allowed is PATH_MAX – currently this is 4096 characters.

It is recommended you keep /lib/firmware paths on your root filesystem, avoid having a separate partition for them in order to avoid possible races with lookups and avoid uses of the custom fallback mechanisms documented below.

Firmware and initramfs

Drivers which are built-in to the kernel should have the firmware integrated also as part of the initramfs used to boot the kernel given that otherwise a race is possible with loading the driver and the real rootfs not yet being available. Stuffing the firmware into initramfs resolves this race issue, however note that using initrd does not suffice to address the same race.

There are circumstances that justify not wanting to include firmware into initramfs, such as dealing with large firmware firmware files for the remote-proc subsystem. For such cases using a userspace fallback mechanism is currently the only viable solution as only userspace can know for sure when the real rootfs is ready and mounted.

49.2.5 Fallback mechanisms

A fallback mechanism is supported to allow to overcome failures to do a direct filesystem lookup on the root filesystem or when the firmware simply cannot be installed for practical reasons on the root filesystem. The kernel configuration options related to supporting the firmware fallback mechanism are:

- CONFIG_FW_LOADER_USER_HELPER: enables building the firmware fallback mechanism. Most distributions enable this option today. If enabled but CONFIG_FW_LOADER_USER_HELPER_FALLBACK is disabled, only the custom fallback mechanism is available and for the `request_firmware_nowait()` call.
- CONFIG_FW_LOADER_USER_HELPER_FALLBACK: force enables each request to enable the kobject uevent fallback mechanism on all firmware API calls except `request_firmware_direct()`. Most distributions disable this option today. The call `request_firmware_nowait()` allows for one alternative fallback mechanism: if this kconfig option is enabled and your second argument to `request_firmware_nowait()`, uevent, is set to false you are informing the kernel that you have a custom fallback mechanism and it will manually load the firmware. Read below for more details.

Note that this means when having this configuration:

CONFIG_FW_LOADER_USER_HELPER=y CONFIG_FW_LOADER_USER_HELPER_FALLBACK=

the kobject uevent fallback mechanism will never take effect even for `request_firmware_nowait()` when uevent is set to true.

Justifying the firmware fallback mechanism

Direct filesystem lookups may fail for a variety of reasons. Known reasons for this are worth itemizing and documenting as it justifies the need for the fallback mechanism:

- Race against access with the root filesystem upon bootup.
- Races upon resume from suspend. This is resolved by the firmware cache, but the firmware cache is only supported if you use uevents, and its not supported for `request_firmware_into_buf()`.
- **Firmware is not accessible through typical means:**
  - It cannot be installed into the root filesystem
  - The firmware provides very unique device specific data tailored for the unit gathered with local information. An example is calibration data for WiFi chipsets for mobile devices. This calibration data is not common to all units, but tailored per unit. Such information may be installed on a separate flash partition other than where the root filesystem is provided.

Types of fallback mechanisms

There are really two fallback mechanisms available using one shared sysfs interface as a loading facility:

- Kobject uevent fallback mechanism
- Custom fallback mechanism

First lets document the shared sysfs loading facility.

Firmware sysfs loading facility

In order to help device drivers upload firmware using a fallback mechanism the firmware infrastructure creates a sysfs interface to enable userspace to load and indicate when firmware is ready. The sysfs directory is created via fw_create_instance(). This call creates a new struct device named after the firmware requested, and establishes it in the device hierarchy by associating the device used to make the request as the device's parent. The sysfs directory's file attributes are defined and controlled through the new device's class (firmware_class) and group (fw_dev_attr_groups). This is actually where the original firmware_class module name came from, given that originally the only firmware loading mechanism available was the mechanism we now use as a fallback mechanism, which registers a struct class firmware_class. Because the attributes exposed are part of the module name, the module name firmware_class cannot be renamed in the future, to ensure backward compatibility with old userspace.

To load firmware using the sysfs interface we expose a loading indicator, and a file upload firmware into:

- /sys/$DEVPATH/loading
- /sys/$DEVPATH/data

To upload firmware you will echo 1 onto the loading file to indicate you are loading firmware. You then write the firmware into the data file, and you notify the kernel the firmware is ready by echo'ing 0 onto the loading file.

The firmware device used to help load firmware using sysfs is only created if direct firmware loading fails and if the fallback mechanism is enabled for your firmware request, this is set up with `firmware_fallback_sysfs()`. It is important to reiterate that no device is created if a direct filesystem lookup succeeded.

Using:

```
echo 1 > /sys/$DEVPATH/loading
```

Will clean any previous partial load at once and make the firmware API return an error. When loading firmware the firmware_class grows a buffer for the firmware in PAGE_SIZE increments to hold the image as it comes in.

firmware_data_read() and firmware_loading_show() are just provided for the test_firmware driver for testing, they are not called in normal use or expected to be used regularly by userspace.

firmware_fallback_sysfs

int **firmware_fallback_sysfs**(struct firmware * fw, const char * name, struct device * device, u32 opt_flags, int ret)
use the fallback mechanism to find firmware

Parameters

struct firmware * fw pointer to firmware image

const char * name name of firmware file to look for

struct device * device device for which firmware is being loaded

u32 opt_flags options to control firmware loading behaviour

int ret return value from direct lookup which triggered the fallback mechanism

Description

This function is called if direct lookup for the firmware failed, it enables a fallback mechanism through userspace by exposing a sysfs loading interface. Userspace is in charge of loading the firmware through the sysfs loading interface. This sysfs fallback mechanism may be disabled completely on a system by setting the proc sysctl value ignore_sysfs_fallback to true. If this is false we check if the internal API caller set the **FW_OPT_NOFALLBACK_SYSFS** flag, if so it would also disable the fallback mechanism. A system may want to enforce the sysfs fallback mechanism at all times, it can do this by setting ignore_sysfs_fallback to false and force_sysfs_fallback to true. Enabling force_sysfs_fallback is functionally equivalent to build a kernel with CONFIG_FW_LOADER_USER_HELPER_FALLBACK.

Firmware kobject uevent fallback mechanism

Since a device is created for the sysfs interface to help load firmware as a fallback mechanism userspace can be informed of the addition of the device by relying on kobject uevents. The addition of the device into the device hierarchy means the fallback mechanism for firmware loading has been initiated. For details of implementation refer to fw_load_sysfs_fallback(), in particular on the use of dev_set_uevent_suppress() and kobject_uevent().

The kernel's kobject uevent mechanism is implemented in lib/kobject_uevent.c, it issues uevents to userspace. As a supplement to kobject uevents Linux distributions could also enable CONFIG_UEVENT_HELPER_PATH, which makes use of core kernel's usermode helper (UMH) functionality to call out to a userspace helper for kobject uevents. In practice though no standard distribution has ever used the CONFIG_UEVENT_HELPER_PATH. If CONFIG_UEVENT_HELPER_PATH is enabled this binary would be called each time kobject_uevent_env() gets called in the kernel for each kobject uevent triggered.

Different implementations have been supported in userspace to take advantage of this fallback mechanism. When firmware loading was only possible using the sysfs mechanism the userspace component "hotplug" provided the functionality of monitoring for kobject events. Historically this was superseded be systemd's udev, however firmware loading support was removed from udev as of systemd commit be2ea723b1d0 ("udev: remove userspace firmware loading support") as of v217 on August, 2014. This means most Linux distributions today are not using or taking advantage of the firmware fallback mechanism provided by kobject uevents. This is specially exacerbated due to the fact that most distributions today disable CONFIG_FW_LOADER_USER_HELPER_FALLBACK.

Refer to do_firmware_uevent() for details of the kobject event variables setup. The variables currently passed to userspace with a "kobject add" event are:

- FIRMWARE=firmware name
- TIMEOUT=timeout value
- ASYNC=whether or not the API request was asynchronous

By default DEVPATH is set by the internal kernel kobject infrastructure. Below is an example simple kobject uevent script:

```
# Both $DEVPATH and $FIRMWARE are already provided in the environment.
MY_FW_DIR=/lib/firmware/
echo 1 > /sys/$DEVPATH/loading
cat $MY_FW_DIR/$FIRMWARE > /sys/$DEVPATH/data
echo 0 > /sys/$DEVPATH/loading
```

Firmware custom fallback mechanism

Users of the `request_firmware_nowait()` call have yet another option available at their disposal: rely on the sysfs fallback mechanism but request that no kobject uevents be issued to userspace. The original logic behind this was that utilities other than udev might be required to lookup firmware in non-traditional paths – paths outside of the listing documented in the section ‘Direct filesystem lookup’. This option is not available to any of the other API calls as uevents are always forced for them.

Since uevents are only meaningful if the fallback mechanism is enabled in your kernel it would seem odd to enable uevents with kernels that do not have the fallback mechanism enabled in their kernels. Unfortunately we also rely on the uevent flag which can be disabled by `request_firmware_nowait()` to also setup the firmware cache for firmware requests. As documented above, the firmware cache is only set up if uevent is enabled for an API call. Although this can disable the firmware cache for `request_firmware_nowait()` calls, users of this API should not use it for the purposes of disabling the cache as that was not the original purpose of the flag. Not setting the uevent flag means you want to opt-in for the firmware fallback mechanism but you want to suppress kobject uevents, as you have a custom solution which will monitor for your device addition into the device hierarchy somehow and load firmware for you through a custom path.

Firmware fallback timeout

The firmware fallback mechanism has a timeout. If firmware is not loaded onto the sysfs interface by the timeout value an error is sent to the driver. By default the timeout is set to 60 seconds if uevents are desirable, otherwise MAX_JIFFY_OFFSET is used (max timeout possible). The logic behind using MAX_JIFFY_OFFSET for non-uevents is that a custom solution will have as much time as it needs to load firmware.

You can customize the firmware timeout by echo’ing your desired timeout into the following file:

- /sys/class/firmware/timeout

If you echo 0 into it means MAX_JIFFY_OFFSET will be used. The data type for the timeout is an int.

EFI embedded firmware fallback mechanism

On some devices the system’s EFI code / ROM may contain an embedded copy of firmware for some of the system’s integrated peripheral devices and the peripheral’s Linux device-driver needs to access this firmware.

Device drivers which need such firmware can use the `firmware_request_platform()` function for this, note that this is a separate fallback mechanism from the other fallback mechanisms and this does not use the sysfs interface.

A device driver which needs this can describe the firmware it needs using an efi_embedded_fw_desc struct:

struct **efi_embedded_fw_desc**
This struct is used by the EFI embedded-fw code to search for embedded firmwares.

Definition

```
struct efi_embedded_fw_desc {
  const char *name;
  u8 prefix[EFI_EMBEDDED_FW_PREFIX_LEN];
  u32 length;
  u8 sha256[32];
};
```

Members

name Name to register the firmware with if found

prefix First 8 bytes of the firmware

length Length of the firmware in bytes including prefix

sha256 SHA256 of the firmware

The EFI embedded-fw code works by scanning all EFI_BOOT_SERVICES_CODE memory segments for an eight byte sequence matching prefix; if the prefix is found it then does a sha256 over length bytes and if that matches makes a copy of length bytes and adds that to its list with found firmwares.

To avoid doing this somewhat expensive scan on all systems, dmi matching is used. Drivers are expected to export a dmi_system_id array, with each entries' driver_data pointing to an efi_embedded_fw_desc.

To register this array with the efi-embedded-fw code, a driver needs to:

1. Always be builtin to the kernel or store the dmi_system_id array in a separate object file which always gets builtin.
2. Add an extern declaration for the dmi_system_id array to include/linux/efi_embedded_fw.h.
3. Add the dmi_system_id array to the embedded_fw_table in drivers/firmware/efi/embedded-firmware.c wrapped in a #ifdef testing that the driver is being builtin.
4. Add "select EFI_EMBEDDED_FIRMWARE if EFI_STUB" to its Kconfig entry.

The `firmware_request_platform()` function will always first try to load firmware with the specified name directly from the disk, so the EFI embedded-fw can always be overridden by placing a file under /lib/firmware.

Note that:

1. The code scanning for EFI embedded-firmware runs near the end of start_kernel(), just before calling rest_init(). For normal drivers and subsystems using subsys_initcall() to register themselves this does not matter. This means that code running earlier cannot use EFI embedded-firmware.
2. At the moment the EFI embedded-fw code assumes that firmwares always start at an offset which is a multiple of 8 bytes, if this is not true for your case send in a patch to fix this.

3. At the moment the EFI embedded-fw code only works on x86 because other archs free EFI_BOOT_SERVICES_CODE before the EFI embedded-fw code gets a chance to scan it.
4. The current brute-force scanning of EFI_BOOT_SERVICES_CODE is an ad-hoc brute-force solution. There has been discussion to use the UEFI Platform Initialization (PI) spec's Firmware Volume protocol. This has been rejected because the FV Protocol relies on internal interfaces of the PI spec, and: 1. The PI spec does not define peripheral firmware at all 2. The internal interfaces of the PI spec do not guarantee any backward compatibility. Any implementation details in FV may be subject to change, and may vary system to system. Supporting the FV Protocol would be difficult as it is purposely ambiguous.

Example how to check for and extract embedded firmware

To check for, for example Silead touchscreen controller embedded firmware, do the following:

1. Boot the system with efi=debug on the kernel commandline
2. cp /sys/kernel/debug/efi/boot_services_code? to your home dir
3. Open the boot_services_code? files in a hex-editor, search for the magic prefix for Silead firmware: F0 00 00 00 02 00 00 00, this gives you the beginning address of the firmware inside the boot_services_code? file.
4. The firmware has a specific pattern, it starts with a 8 byte page-address, typically F0 00 00 00 02 00 00 00 for the first page followed by 32-bit word-address + 32-bit value pairs. With the word-address incrementing 4 bytes (1 word) for each pair until a page is complete. A complete page is followed by a new page-address, followed by more word + value pairs. This leads to a very distinct pattern. Scroll down until this pattern stops, this gives you the end of the firmware inside the boot_services_code? file.
5. "dd if=boot_services_code? of=firmware bs=1 skip=<begin-addr> count=<len>" will extract the firmware for you. Inspect the firmware file in a hexeditor to make sure you got the dd parameters correct.
6. Copy it to /lib/firmware under the expected name to test it.
7. If the extracted firmware works, you can use the found info to fill an efi_embedded_fw_desc struct to describe it, run "sha256sum firmware" to get the sha256sum to put in the sha256 field.

49.2.6 Firmware lookup order

Different functionality is available to enable firmware to be found. Below is chronological order of how firmware will be looked for once a driver issues a firmware API call.

- The “Built-in firmware” is checked first, if the firmware is present we return it immediately
- The “Firmware cache” is looked at next. If the firmware is found we return it immediately
- The “Direct filesystem lookup” is performed next, if found we return it immediately
- The “Platform firmware fallback” is performed next, but only when `firmware_request_platform()` is used, if found we return it immediately
- If no firmware has been found and the fallback mechanism was enabled the sysfs interface is created. After this either a kobject uevent is issued or the custom firmware loading is relied upon for firmware loading up to the timeout value.

49.3 UEFI Support

49.3.1 UEFI stub library functions

efi_status_t **efi_get_memory_map**(struct efi_boot_memmap * map)
get memory map

Parameters

struct efi_boot_memmap * map on return pointer to memory map

Description

Retrieve the UEFI memory map. The allocated memory leaves room for up to EFI_MMAP_NR_SLACK_SLOTS additional memory map entries.

Return

status code

efi_status_t **efi_allocate_pages**(unsigned long size, unsigned long * addr, unsigned long max)
Allocate memory pages

Parameters

unsigned long size minimum number of bytes to allocate

unsigned long * addr On return the address of the first allocated page. The first allocated page has alignment EFI_ALLOC_ALIGN which is an architecture dependent multiple of the page size.

unsigned long max the address that the last allocated memory page shall not exceed

Description

Allocate pages as EFI_LOADER_DATA. The allocated pages are aligned according to EFI_ALLOC_ALIGN. The last allocated page will not exceed the address given by **max**.

Return

status code

void **efi_free**(unsigned long size, unsigned long addr)
free memory pages

Parameters

unsigned long size size of the memory area to free in bytes

unsigned long addr start of the memory area to free (must be EFI_PAGE_SIZE aligned)

Description

size is rounded up to a multiple of EFI_ALLOC_ALIGN which is an architecture specific multiple of EFI_PAGE_SIZE. So this function should only be used to return pages allocated with `efi_allocate_pages()` or efi_low_alloc_above().

49.4 request_firmware API

You would typically load firmware and then load it into your device somehow. The typical firmware work flow is reflected below:

```
if(request_firmware(&fw_entry, $FIRMWARE, device) == 0)
       copy_fw_to_device(fw_entry->data, fw_entry->size);
release_firmware(fw_entry);
```

49.4.1 Synchronous firmware requests

Synchronous firmware requests will wait until the firmware is found or until an error is returned.

request_firmware

int **request_firmware**(const struct firmware ** firmware_p, const char * name, struct device * device)
send firmware request and wait for it

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded

firmware_p will be used to return a firmware image by the name of **name** for device **device**.

Should be called from user context where sleeping is allowed.

name will be used as $FIRMWARE in the uevent environment and should be distinctive enough not to be confused with any other firmware image for this or any other device.

Caller must hold the reference count of **device**.

The function can be called safely inside device's suspend and resume callback.

firmware_request_nowarn

int **firmware_request_nowarn**(const struct firmware ** firmware, const char * name, struct device * device)
request for an optional fw module

Parameters

const struct firmware ** firmware pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded

Description

This function is similar in behaviour to `request_firmware()`, except it doesn't produce warning messages when the file is not found. The sysfs fallback mechanism is enabled if direct filesystem lookup fails, however, however failures to find the firmware file with it are still suppressed. It is therefore up to the driver to check for the return value of this call and to decide when to inform the users of errors.

firmware_request_platform

int **firmware_request_platform**(const struct firmware ** firmware, const char * name, struct device * device)
request firmware with platform-fw fallback

Parameters

const struct firmware ** firmware pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded

Description

This function is similar in behaviour to request_firmware, except that if direct filesystem lookup fails, it will fallback to looking for a copy of the requested firmware embedded in the platform's main (e.g. UEFI) firmware.

request_firmware_direct

int **request_firmware_direct**(const struct firmware ** firmware_p, const char * name, struct device * device)
load firmware directly without usermode helper

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded

Description

This function works pretty much like `request_firmware()`, but this doesn't fall back to usermode helper even if the firmware couldn't be loaded directly from fs. Hence it's useful for loading optional firmwares, which aren't always present, without extra long timeouts of udev.

request_firmware_into_buf

int **request_firmware_into_buf**(const struct firmware ** firmware_p, const char * name, struct device * device, void * buf, size_t size)
load firmware into a previously allocated buffer

Parameters

const struct firmware ** firmware_p pointer to firmware image

const char * name name of firmware file

struct device * device device for which firmware is being loaded and DMA region allocated

void * buf address of buffer to load firmware into

size_t size size of buffer

Description

This function works pretty much like `request_firmware()`, but it doesn't allocate a buffer to hold the firmware data. Instead, the firmware is loaded directly into the buffer pointed to by **buf** and the **firmware_p** data member is pointed at **buf**.

This function doesn't cache firmware either.

49.4.2 Asynchronous firmware requests

Asynchronous firmware requests allow driver code to not have to wait until the firmware or an error is returned. Function callbacks are provided so that when the firmware or an error is found the driver is informed through the callback. `request_firmware_nowait()` cannot be called in atomic contexts.

request_firmware_nowait

int **request_firmware_nowait**(struct module * module, bool uevent, const char * name, struct device * device, gfp_t gfp, void * context, void (*cont)(const struct firmware *fw, void *context))

asynchronous version of request_firmware

Parameters

struct module * module module requesting the firmware

bool uevent sends uevent to copy the firmware image if this flag is non-zero else the firmware copy must be done manually.

const char * name name of firmware file

struct device * device device for which firmware is being loaded

gfp_t gfp allocation flags

void * context will be passed over to **cont**, and **fw** may be `NULL` if firmware request fails.

void (*)(const struct firmware *fw, void *context) cont function will be called asynchronously when the firmware request is over.

Caller must hold the reference count of **device**.

Asynchronous variant of request_firmware() for user contexts:

- sleep for as small periods as possible since it may increase kernel boot time of built-in device drivers requesting firmware in their ->probe() methods, if **gfp** is GFP_KERNEL.
- can't sleep at all if **gfp** is GFP_ATOMIC.

49.4.3 Special optimizations on reboot

Some devices have an optimization in place to enable the firmware to be retained during system reboot. When such optimizations are used the driver author must ensure the firmware is still available on resume from suspend, this can be done with `firmware_request_cache()` instead of requesting for the firmware to be loaded.

firmware_request_cache()

int **firmware_request_cache**(struct device * device, const char * name)
cache firmware for suspend so resume can use it

Parameters

struct device * device device for which firmware should be cached for

const char * name name of firmware file

Description

There are some devices with an optimization that enables the device to not require loading firmware on system reboot. This optimization may still require the firmware present on resume from suspend. This routine can be used to ensure the firmware is present on resume from suspend in these situations. This helper is not compatible with drivers which use `request_firmware_into_buf()` or `request_firmware_nowait()` with no uevent set.

49.4.4 request firmware API expected driver use

Once an API call returns you process the firmware and then release the firmware. For example if you used `request_firmware()` and it returns, the driver has the firmware image accessible in fw_entry->{data,size}. If something went wrong `request_firmware()` returns non-zero and fw_entry is set to NULL. Once your driver is done with processing the firmware it can call call release_firmware(fw_entry) to release the firmware image and any related resource.

49.5 Other Firmware Interfaces

49.5.1 DMI Interfaces

int **dmi_check_system**(const struct dmi_system_id * list)
check system DMI data

Parameters

const struct dmi_system_id * list array of dmi_system_id structures to match against All non-null elements of the list must match their slot's (field index's) data (i.e., each list string must be a substring of the specified DMI slot's string data) to be considered a successful match.

Walk the blacklist table running matching functions until someone returns non zero or we hit the end. Callback function is called for each successful match. Returns the number of matches.

dmi_setup must be called before this function is called.

const struct dmi_system_id * **dmi_first_match**(const struct dmi_system_id * list)
find dmi_system_id structure matching system DMI data

Parameters

const struct dmi_system_id * list array of dmi_system_id structures to match against All non-null elements of the list must match their slot's (field index's) data (i.e., each list string must be a substring of the specified DMI slot's string data) to be considered a successful match.

Walk the blacklist table until the first match is found. Return the pointer to the matching entry or NULL if there's no match.

dmi_setup must be called before this function is called.

const char * **dmi_get_system_info**(int field)
return DMI data value

Parameters

int field data index (see enum dmi_field)

Returns one DMI data value, can be used to perform complex DMI data checks.

int **dmi_name_in_vendors**(const char * str)
Check if string is in the DMI system or board vendor name

Parameters

const char * str Case sensitive Name

const struct dmi_device * **dmi_find_device**(int type, const char * name, const struct dmi_device * from)
find onboard device by type/name

Parameters

int type device type or `DMI_DEV_TYPE_ANY` to match all device types

const char * name device name string or `NULL` to match all

const struct dmi_device * from previous device found in search, or `NULL` for new search.

Iterates through the list of known onboard devices. If a device is found with a matching **type** and **name**, a pointer to its device structure is returned. Otherwise, `NULL` is returned. A new search is initiated by passing `NULL` as the **from** argument. If **from** is not `NULL`, searches continue from next device.

bool **dmi_get_date**(int field, int * yearp, int * monthp, int * dayp)
parse a DMI date

Parameters

int field data index (see enum dmi_field)

int * yearp optional out parameter for the year

int * monthp optional out parameter for the month

int * dayp optional out parameter for the day

The date field is assumed to be in the form resembling [mm[/dd]]/yy[yy] and the result is stored in the out parameters any or all of which can be omitted.

If the field doesn't exist, all out parameters are set to zero and false is returned. Otherwise, true is returned with any invalid part of date set to zero.

On return, year, month and day are guaranteed to be in the range of [0,9999], [0,12] and [0,31] respectively.

int **dmi_get_bios_year**(void)
get a year out of DMI_BIOS_DATE field

Parameters

void no arguments

Description

Returns year on success, -ENXIO if DMI is not selected, or a different negative error code if DMI field is not present or not parseable.

int **dmi_walk**(void (*decode)(const struct dmi_header *, void *), void * private_data)
Walk the DMI table and get called back for every record

Parameters

void (*)(const struct dmi_header *, void *) decode Callback function

void * private_data Private data to be passed to the callback function

Returns 0 on success, -ENXIO if DMI is not selected or not present, or a different negative error code if DMI walking fails.

bool **dmi_match**(enum dmi_field f, const char * str)
compare a string to the dmi field (if exists)

Parameters

enum dmi_field f DMI field identifier

const char * str string to compare the DMI field to

Description

Returns true if the requested field equals to the str (including NULL).

u8 **dmi_memdev_type**(u16 handle)
get the memory type

Parameters

u16 handle DMI structure handle

Description

Return the DMI memory type of the module in the slot associated with the given DMI handle, or 0x0 if no such DMI handle exists.

u16 **dmi_memdev_handle**(int slot)
get the DMI handle of a memory slot

Parameters

int slot slot number

Return the DMI handle associated with a given memory slot, or `0xFFFF` if there is no such slot.

49.5.2 EDD Interfaces

ssize_t **edd_show_raw_data**(struct edd_device * edev, char * buf)
copies raw data to buffer for userspace to parse

Parameters

struct edd_device * edev target edd_device

char * buf output buffer

Return

number of bytes written, or -EINVAL on failure

void **edd_release**(struct kobject * kobj)
free edd structure

Parameters

struct kobject * kobj kobject of edd structure

This is called when the refcount of the edd structure reaches 0. This should happen right after we unregister, but just in case, we use the release callback anyway.

int **edd_dev_is_type**(struct edd_device * edev, const char * type)
is this EDD device a 'type' device?

Parameters

struct edd_device * edev target edd_device

const char * type a host bus or interface identifier string per the EDD spec

Description

Returns 1 (TRUE) if it is a 'type' device, 0 otherwise.

struct pci_dev * **edd_get_pci_dev**(struct edd_device * edev)
finds pci_dev that matches edev

Parameters

struct edd_device * edev edd_device

Description

Returns pci_dev if found, or NULL

int **edd_init**(void)
creates sysfs tree of EDD data

Parameters

void no arguments

49.5.3 Intel Stratix10 SoC Service Layer

Some features of the Intel Stratix10 SoC require a level of privilege higher than the kernel is granted. Such secure features include FPGA programming. In terms of the ARMv8 architecture, the kernel runs at Exception Level 1 (EL1), access to the features requires Exception Level 3 (EL3).

The Intel Stratix10 SoC service layer provides an in kernel API for drivers to request access to the secure features. The requests are queued and processed one by one. ARM's SMCCC is used to pass the execution of the requests on to a secure monitor (EL3).

enum **stratix10_svc_command_code**
supported service commands

Constants

COMMAND_NOOP do 'dummy' request for integration/debug/trouble-shooting

COMMAND_RECONFIG ask for FPGA configuration preparation, return status is SVC_STATUS_OK

COMMAND_RECONFIG_DATA_SUBMIT submit buffer(s) of bit-stream data for the FPGA configuration, return status is SVC_STATUS_SUBMITTED or SVC_STATUS_ERROR

COMMAND_RECONFIG_DATA_CLAIM check the status of the configuration, return status is SVC_STATUS_COMPLETED, or SVC_STATUS_BUSY, or SVC_STATUS_ERROR

COMMAND_RECONFIG_STATUS check the status of the configuration, return status is SVC_STATUS_COMPLETED, or SVC_STATUS_BUSY, or SVC_STATUS_ERROR

COMMAND_RSU_STATUS request remote system update boot log, return status is log data or SVC_STATUS_RSU_ERROR

COMMAND_RSU_UPDATE set the offset of the bitstream to boot after reboot, return status is SVC_STATUS_OK or SVC_STATUS_ERROR

COMMAND_RSU_NOTIFY report the status of hard processor system software to firmware, return status is SVC_STATUS_OK or SVC_STATUS_ERROR

COMMAND_RSU_RETRY query firmware for the current image's retry counter, return status is SVC_STATUS_OK or SVC_STATUS_ERROR

struct **stratix10_svc_client_msg**
message sent by client to service

Definition

```
struct stratix10_svc_client_msg {
  void *payload;
  size_t payload_length;
  enum stratix10_svc_command_code command;
  u64 arg[3];
};
```

Members

payload starting address of data need be processed

payload_length data size in bytes

command service command

arg args to be passed via registers and not physically mapped buffers

struct **stratix10_svc_command_config_type**
config type

Definition

```
struct stratix10_svc_command_config_type {
  u32 flags;
};
```

Members

flags flag bit for the type of FPGA configuration

struct **stratix10_svc_cb_data**
callback data structure from service layer

Definition

```
struct stratix10_svc_cb_data {
  u32 status;
  void *kaddr1;
  void *kaddr2;
  void *kaddr3;
};
```

Members

status the status of sent command

kaddr1 address of 1st completed data block

kaddr2 address of 2nd completed data block

kaddr3 address of 3rd completed data block

struct **stratix10_svc_client**
service client structure

Definition

```
struct stratix10_svc_client {
  struct device *dev;
  void (*receive_cb)(struct stratix10_svc_client *client, struct stratix10_
↪svc_cb_data *cb_data);
  void *priv;
};
```

Members

dev the client device

receive_cb callback to provide service client the received data

priv client private data

struct stratix10_svc_chan * **stratix10_svc_request_channel_byname**(struct stratix10_svc_client * client, const char * name)
request a service channel

Parameters

struct stratix10_svc_client * client pointer to service client

const char * name service client name

Description

This function is used by service client to request a service channel.

Return

a pointer to channel assigned to the client on success, or ERR_PTR() on error.

void **stratix10_svc_free_channel**(struct stratix10_svc_chan * chan)
free service channel

Parameters

struct stratix10_svc_chan * chan service channel to be freed

Description

This function is used by service client to free a service channel.

int **stratix10_svc_send**(struct stratix10_svc_chan * chan, void * msg)
send a message data to the remote

Parameters

struct stratix10_svc_chan * chan service channel assigned to the client

void * msg message data to be sent, in the format of "struct stratix10_svc_client_msg"

Description

This function is used by service client to add a message to the service layer driver's queue for being sent to the secure world.

Return

0 for success, -ENOMEM or -ENOBUFS on error.

void **stratix10_svc_done**(struct stratix10_svc_chan * chan)
complete service request transactions

Parameters

struct stratix10_svc_chan * chan service channel assigned to the client

Description

This function should be called when client has finished its request or there is an error in the request process. It allows the service layer to stop the running thread to have maximize savings in kernel resources.

void * **stratix10_svc_allocate_memory**(struct stratix10_svc_chan * chan, size_t size)

allocate memory

Parameters

struct stratix10_svc_chan * chan service channel assigned to the client

size_t size memory size requested by a specific service client

Description

Service layer allocates the requested number of bytes buffer from the memory pool, service client uses this function to get allocated buffers.

Return

address of allocated memory on success, or ERR_PTR() on error.

void **stratix10_svc_free_memory**(struct stratix10_svc_chan * chan, void * kaddr)

free allocated memory

Parameters

struct stratix10_svc_chan * chan service channel assigned to the client

void * kaddr memory to be freed

Description

This function is used by service client to free allocated buffers.

CHAPTER

FIFTY

PINCTRL (PIN CONTROL) SUBSYSTEM

This document outlines the pin control subsystem in Linux

This subsystem deals with:

- Enumerating and naming controllable pins
- Multiplexing of pins, pads, fingers (etc) see below for details
- Configuration of pins, pads, fingers (etc), such as software-controlled biasing and driving mode specific pins, such as pull-up/down, open drain, load capacitance etc.

50.1 Top-level interface

Definition of PIN CONTROLLER:

- A pin controller is a piece of hardware, usually a set of registers, that can control PINs. It may be able to multiplex, bias, set load capacitance, set drive strength, etc. for individual pins or groups of pins.

Definition of PIN:

- PINS are equal to pads, fingers, balls or whatever packaging input or output line you want to control and these are denoted by unsigned integers in the range 0..maxpin. This numberspace is local to each PIN CONTROLLER, so there may be several such number spaces in a system. This pin space may be sparse - i.e. there may be gaps in the space with numbers where no pin exists.

When a PIN CONTROLLER is instantiated, it will register a descriptor to the pin control framework, and this descriptor contains an array of pin descriptors describing the pins handled by this specific pin controller.

Here is an example of a PGA (Pin Grid Array) chip seen from underneath:

```
     A   B   C   D   E   F   G   H

8    o   o   o   o   o   o   o   o

7    o   o   o   o   o   o   o   o

6    o   o   o   o   o   o   o   o
```

(continues on next page)

(continued from previous page)

```
5    o   o   o   o   o   o   o   o

4    o   o   o   o   o   o   o   o

3    o   o   o   o   o   o   o   o

2    o   o   o   o   o   o   o   o

1    o   o   o   o   o   o   o   o
```

To register a pin controller and name all the pins on this package we can do this in our driver:

```
#include <linux/pinctrl/pinctrl.h>

const struct pinctrl_pin_desc foo_pins[] = {
        PINCTRL_PIN(0, "A8"),
        PINCTRL_PIN(1, "B8"),
        PINCTRL_PIN(2, "C8"),
        ...
        PINCTRL_PIN(61, "F1"),
        PINCTRL_PIN(62, "G1"),
        PINCTRL_PIN(63, "H1"),
};

static struct pinctrl_desc foo_desc = {
        .name = "foo",
        .pins = foo_pins,
        .npins = ARRAY_SIZE(foo_pins),
        .owner = THIS_MODULE,
};

int __init foo_probe(void)
{
        int error;

        struct pinctrl_dev *pctl;

        error = pinctrl_register_and_init(&foo_desc, <PARENT>,
                                          NULL, &pctl);
        if (error)
                return error;

        return pinctrl_enable(pctl);
}
```

To enable the pinctrl subsystem and the subgroups for PINMUX and PINCONF and selected drivers, you need to select them from your machine's Kconfig entry, since these are so tightly integrated with the machines they are used on. See for example arch/arm/mach-u300/Kconfig for an example.

Pins usually have fancier names than this. You can find these in the datasheet for your chip. Notice that the core pinctrl.h file provides a fancy macro called PINCTRL_PIN() to create the struct entries. As you can see I enumerated the pins from 0 in the upper left corner to 63 in the lower right corner. This enumeration was arbitrarily chosen, in practice you need to think through your numbering system

so that it matches the layout of registers and such things in your driver, or the code may become complicated. You must also consider matching of offsets to the GPIO ranges that may be handled by the pin controller.

For a padring with 467 pads, as opposed to actual pins, I used an enumeration like this, walking around the edge of the chip, which seems to be industry standard too (all these pads had names, too):

```
  0 ..... 104
466        105
  .         .
  .         .
358        224
 357 .... 225
```

50.2 Pin groups

Many controllers need to deal with groups of pins, so the pin controller subsystem has a mechanism for enumerating groups of pins and retrieving the actual enumerated pins that are part of a certain group.

For example, say that we have a group of pins dealing with an SPI interface on { 0, 8, 16, 24 }, and a group of pins dealing with an I2C interface on pins on { 24, 25 }.

These two groups are presented to the pin control subsystem by implementing some generic pinctrl_ops like this:

```
#include <linux/pinctrl/pinctrl.h>

struct foo_group {
        const char *name;
        const unsigned int *pins;
        const unsigned num_pins;
};

static const unsigned int spi0_pins[] = { 0, 8, 16, 24 };
static const unsigned int i2c0_pins[] = { 24, 25 };

static const struct foo_group foo_groups[] = {
        {
                .name = "spi0_grp",
                .pins = spi0_pins,
                .num_pins = ARRAY_SIZE(spi0_pins),
        },
        {
                .name = "i2c0_grp",
                .pins = i2c0_pins,
                .num_pins = ARRAY_SIZE(i2c0_pins),
        },
};


static int foo_get_groups_count(struct pinctrl_dev *pctldev)
```

(continues on next page)

(continued from previous page)

```
{
        return ARRAY_SIZE(foo_groups);
}

static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
                                      unsigned selector)
{
        return foo_groups[selector].name;
}

static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned␣
↪selector,
                              const unsigned **pins,
                              unsigned *num_pins)
{
        *pins = (unsigned *) foo_groups[selector].pins;
        *num_pins = foo_groups[selector].num_pins;
        return 0;
}

static struct pinctrl_ops foo_pctrl_ops = {
        .get_groups_count = foo_get_groups_count,
        .get_group_name = foo_get_group_name,
        .get_group_pins = foo_get_group_pins,
};


static struct pinctrl_desc foo_desc = {
...
.pctlops = &foo_pctrl_ops,
};
```

The pin control subsystem will call the .get_groups_count() function to determine the total number of legal selectors, then it will call the other functions to retrieve the name and pins of the group. Maintaining the data structure of the groups is up to the driver, this is just a simple example - in practice you may need more entries in your group structure, for example specific register ranges associated with each group and so on.

50.3 Pin configuration

Pins can sometimes be software-configured in various ways, mostly related to their electronic properties when used as inputs or outputs. For example you may be able to make an output pin high impedance, or "tristate" meaning it is effectively disconnected. You may be able to connect an input pin to VDD or GND using a certain resistor value - pull up and pull down - so that the pin has a stable value when nothing is driving the rail it is connected to, or when it's unconnected.

Pin configuration can be programmed by adding configuration entries into the mapping table; see section "Board/machine configuration" below.

The format and meaning of the configuration parameter, PLATFORM_X_PULL_UP above, is entirely defined by the pin controller driver.

The pin configuration driver implements callbacks for changing pin configuration in the pin controller ops like this:

```
#include <linux/pinctrl/pinctrl.h>
#include <linux/pinctrl/pinconf.h>
#include "platform_x_pindefs.h"

static int foo_pin_config_get(struct pinctrl_dev *pctldev,
                unsigned offset,
                unsigned long *config)
{
        struct my_conftype conf;

        ... Find setting for pin @ offset ...

        *config = (unsigned long) conf;
}

static int foo_pin_config_set(struct pinctrl_dev *pctldev,
                unsigned offset,
                unsigned long config)
{
        struct my_conftype *conf = (struct my_conftype *) config;

        switch (conf) {
                case PLATFORM_X_PULL_UP:
                ...
                }
        }
}

static int foo_pin_config_group_get (struct pinctrl_dev *pctldev,
                unsigned selector,
                unsigned long *config)
{
        ...
}

static int foo_pin_config_group_set (struct pinctrl_dev *pctldev,
                unsigned selector,
                unsigned long config)
{
        ...
}

static struct pinconf_ops foo_pconf_ops = {
        .pin_config_get = foo_pin_config_get,
        .pin_config_set = foo_pin_config_set,
        .pin_config_group_get = foo_pin_config_group_get,
        .pin_config_group_set = foo_pin_config_group_set,
};

/* Pin config operations are handled by some pin controller */
static struct pinctrl_desc foo_desc = {
        ...
        .confops = &foo_pconf_ops,
};
```

50.4 Interaction with the GPIO subsystem

The GPIO drivers may want to perform operations of various types on the same physical pins that are also registered as pin controller pins.

First and foremost, the two subsystems can be used as completely orthogonal, see the section named "pin control requests from drivers" and "drivers needing both pin control and GPIOs" below for details. But in some situations a cross-subsystem mapping between pins and GPIOs is needed.

Since the pin controller subsystem has its pinspace local to the pin controller we need a mapping so that the pin control subsystem can figure out which pin controller handles control of a certain GPIO pin. Since a single pin controller may be muxing several GPIO ranges (typically SoCs that have one set of pins, but internally several GPIO silicon blocks, each modelled as a struct gpio_chip) any number of GPIO ranges can be added to a pin controller instance like this:

```
struct gpio_chip chip_a;
struct gpio_chip chip_b;

static struct pinctrl_gpio_range gpio_range_a = {
        .name = "chip a",
        .id = 0,
        .base = 32,
        .pin_base = 32,
        .npins = 16,
        .gc = &chip_a;
};

static struct pinctrl_gpio_range gpio_range_b = {
        .name = "chip b",
        .id = 0,
        .base = 48,
        .pin_base = 64,
        .npins = 8,
        .gc = &chip_b;
};

{
        struct pinctrl_dev *pctl;
        ...
        pinctrl_add_gpio_range(pctl, &gpio_range_a);
        pinctrl_add_gpio_range(pctl, &gpio_range_b);
}
```

So this complex system has one pin controller handling two different GPIO chips. "chip a" has 16 pins and "chip b" has 8 pins. The "chip a" and "chip b" have different .pin_base, which means a start pin number of the GPIO range.

The GPIO range of "chip a" starts from the GPIO base of 32 and actual pin range also starts from 32. However "chip b" has different starting offset for the GPIO range and pin range. The GPIO range of "chip b" starts from GPIO number 48, while the pin range of "chip b" starts from 64.

We can convert a gpio number to actual pin number using this "pin_base". They are mapped in the global GPIO pin space at:

chip a:

- GPIO range : [32 .. 47]
- pin range : [32 .. 47]

chip b:

- GPIO range : [48 .. 55]
- pin range : [64 .. 71]

The above examples assume the mapping between the GPIOs and pins is linear. If the mapping is sparse or haphazard, an array of arbitrary pin numbers can be encoded in the range like this:

```
static const unsigned range_pins[] = { 14, 1, 22, 17, 10, 8, 6, 2 };

static struct pinctrl_gpio_range gpio_range = {
        .name = "chip",
        .id = 0,
        .base = 32,
        .pins = &range_pins,
        .npins = ARRAY_SIZE(range_pins),
        .gc = &chip;
};
```

In this case the pin_base property will be ignored. If the name of a pin group is known, the pins and npins elements of the above structure can be initialised using the function pinctrl_get_group_pins(), e.g. for pin group “foo”:

```
pinctrl_get_group_pins(pctl, "foo", &gpio_range.pins,
                       &gpio_range.npins);
```

When GPIO-specific functions in the pin control subsystem are called, these ranges will be used to look up the appropriate pin controller by inspecting and matching the pin to the pin ranges across all controllers. When a pin controller handling the matching range is found, GPIO-specific functions will be called on that specific pin controller.

For all functionalities dealing with pin biasing, pin muxing etc, the pin controller subsystem will look up the corresponding pin number from the passed in gpio number, and use the range’s internals to retrieve a pin number. After that, the subsystem passes it on to the pin control driver, so the driver will get a pin number into its handled number range. Further it is also passed the range ID value, so that the pin controller knows which range it should deal with.

Calling pinctrl_add_gpio_range from pinctrl driver is DEPRECATED. Please see section 2.1 of Documentation/devicetree/bindings/gpio/gpio.txt on how to bind pinctrl and gpio drivers.

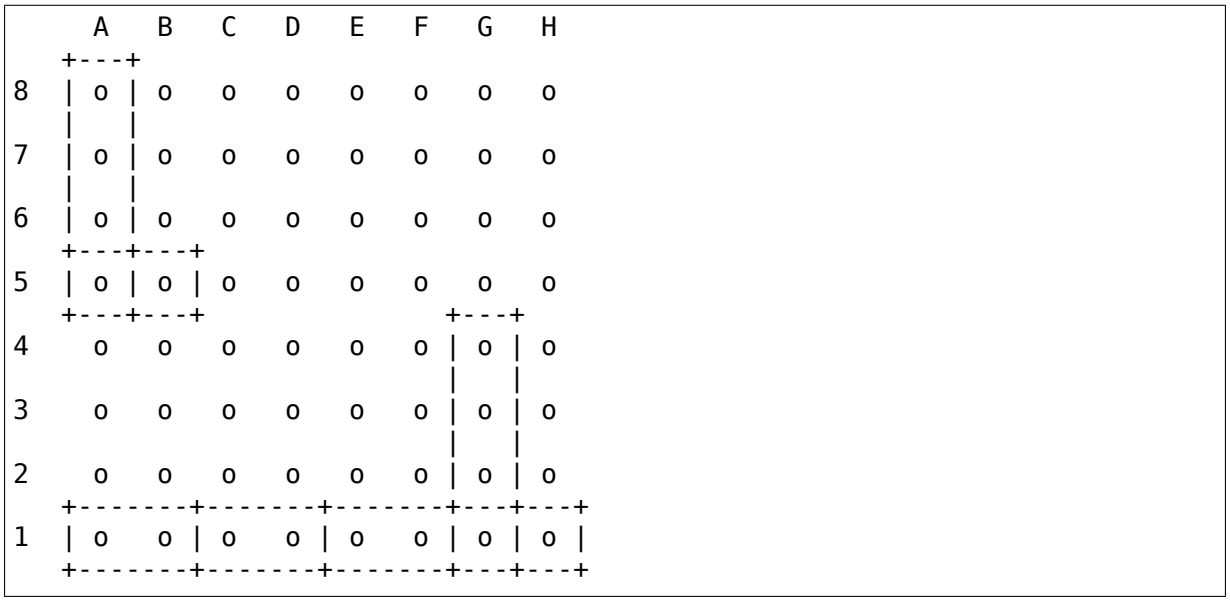
50.5 PINMUX interfaces

These calls use the pinmux_* naming prefix. No other calls should use that prefix.

50.6 What is pinmuxing?

PINMUX, also known as padmux, ballmux, alternate functions or mission modes is a way for chip vendors producing some kind of electrical packages to use a certain physical pin (ball, pad, finger, etc) for multiple mutually exclusive functions, depending on the application. By "application" in this context we usually mean a way of soldering or wiring the package into an electronic system, even though the framework makes it possible to also change the function at runtime.

Here is an example of a PGA (Pin Grid Array) chip seen from underneath:



This is not tetris. The game to think of is chess. Not all PGA/BGA packages are chessboard-like, big ones have "holes" in some arrangement according to different design patterns, but we' re using this as a simple example. Of the pins you see some will be taken by things like a few VCC and GND to feed power to the chip, and quite a few will be taken by large ports like an external memory interface. The remaining pins will often be subject to pin multiplexing.

The example 8x8 PGA package above will have pin numbers 0 through 63 assigned to its physical pins. It will name the pins { A1, A2, A3 ···H6, H7, H8 } using pinctrl_register_pins() and a suitable data set as shown earlier.

In this 8x8 BGA package the pins { A8, A7, A6, A5 } can be used as an SPI port (these are four pins: CLK, RXD, TXD, FRM). In that case, pin B5 can be used as some general-purpose GPIO pin. However, in another setting, pins { A5, B5 } can be used as an I2C port (these are just two pins: SCL, SDA). Needless to say, we cannot use the SPI port and I2C port at the same time. However in the inside of the package the silicon performing the SPI logic can alternatively be routed out on pins { G4, G3, G2, G1 }.

On the bottom row at { A1, B1, C1, D1, E1, F1, G1, H1 } we have something special - it's an external MMC bus that can be 2, 4 or 8 bits wide, and it will consume 2, 4 or 8 pins respectively, so either { A1, B1 } are taken or { A1, B1, C1, D1 } or all of them. If we use all 8 bits, we cannot use the SPI port on pins { G4, G3, G2, G1 } of course.

This way the silicon blocks present inside the chip can be multiplexed "muxed" out on different pin ranges. Often contemporary SoC (systems on chip) will contain several I2C, SPI, SDIO/MMC, etc silicon blocks that can be routed to different pins by pinmux settings.

Since general-purpose I/O pins (GPIO) are typically always in shortage, it is common to be able to use almost any pin as a GPIO pin if it is not currently in use by some other I/O port.

50.7 Pinmux conventions

The purpose of the pinmux functionality in the pin controller subsystem is to abstract and provide pinmux settings to the devices you choose to instantiate in your machine configuration. It is inspired by the clk, GPIO and regulator subsystems, so devices will request their mux setting, but it's also possible to request a single pin for e.g. GPIO.

Definitions:

- FUNCTIONS can be switched in and out by a driver residing with the pin control subsystem in the drivers/pinctrl/* directory of the kernel. The pin control driver knows the possible functions. In the example above you can identify three pinmux functions, one for spi, one for i2c and one for mmc.
- FUNCTIONS are assumed to be enumerable from zero in a one-dimensional array. In this case the array could be something like: { spi0, i2c0, mmc0 } for the three available functions.
- FUNCTIONS have PIN GROUPS as defined on the generic level - so a certain function is always associated with a certain set of pin groups, could be just a single one, but could also be many. In the example above the function i2c is associated with the pins { A5, B5 }, enumerated as { 24, 25 } in the controller pin space.

  The Function spi is associated with pin groups { A8, A7, A6, A5 } and { G4, G3, G2, G1 }, which are enumerated as { 0, 8, 16, 24 } and { 38, 46, 54, 62 } respectively.

  Group names must be unique per pin controller, no two groups on the same controller may have the same name.
- The combination of a FUNCTION and a PIN GROUP determine a certain function for a certain set of pins. The knowledge of the functions and pin groups and their machine-specific particulars are kept inside the pinmux driver, from the outside only the enumerators are known, and the driver core can request:
  - The name of a function with a certain selector (>= 0)
  - A list of groups associated with a certain function

- That a certain group in that list to be activated for a certain function

As already described above, pin groups are in turn self-descriptive, so the core will retrieve the actual pin range in a certain group from the driver.

- FUNCTIONS and GROUPS on a certain PIN CONTROLLER are MAPPED to a certain device by the board file, device tree or similar machine setup configuration mechanism, similar to how regulators are connected to devices, usually by name. Defining a pin controller, function and group thus uniquely identify the set of pins to be used by a certain device. (If only one possible group of pins is available for the function, no group name need to be supplied - the core will simply select the first and only group available.)

  In the example case we can define that this particular machine shall use device spi0 with pinmux function fspi0 group gspi0 and i2c0 on function fi2c0 group gi2c0, on the primary pin controller, we get mappings like these:

```
{
        {"map-spi0", spi0, pinctrl0, fspi0, gspi0},
        {"map-i2c0", i2c0, pinctrl0, fi2c0, gi2c0}
}
```

  Every map must be assigned a state name, pin controller, device and function. The group is not compulsory - if it is omitted the first group presented by the driver as applicable for the function will be selected, which is useful for simple cases.

  It is possible to map several groups to the same combination of device, pin controller and function. This is for cases where a certain function on a certain pin controller may use different sets of pins in different configurations.

- PINS for a certain FUNCTION using a certain PIN GROUP on a certain PIN CONTROLLER are provided on a first-come first-serve basis, so if some other device mux setting or GPIO pin request has already taken your physical pin, you will be denied the use of it. To get (activate) a new setting, the old one has to be put (deactivated) first.

Sometimes the documentation and hardware registers will be oriented around pads (or “fingers”) rather than pins - these are the soldering surfaces on the silicon inside the package, and may or may not match the actual number of pins/balls underneath the capsule. Pick some enumeration that makes sense to you. Define enumerators only for the pins you can control if that makes sense.

Assumptions:

We assume that the number of possible function maps to pin groups is limited by the hardware. I.e. we assume that there is no system where any function can be mapped to any pin, like in a phone exchange. So the available pin groups for a certain function will be limited to a few choices (say up to eight or so), not hundreds or any amount of choices. This is the characteristic we have found by inspecting available pinmux hardware, and a necessary assumption since we expect pinmux drivers to present all possible function vs pin group mappings to the subsystem.

50.8 Pinmux drivers

The pinmux core takes care of preventing conflicts on pins and calling the pin controller driver to execute different settings.

It is the responsibility of the pinmux driver to impose further restrictions (say for example infer electronic limitations due to load, etc.) to determine whether or not the requested function can actually be allowed, and in case it is possible to perform the requested mux setting, poke the hardware so that this happens.

Pinmux drivers are required to supply a few callback functions, some are optional. Usually the set_mux() function is implemented, writing values into some certain registers to activate a certain mux setting for a certain pin.

A simple driver for the above example will work by setting bits 0, 1, 2, 3 or 4 into some register named MUX to select a certain function with a certain group of pins would work something like this:

```
#include <linux/pinctrl/pinctrl.h>
#include <linux/pinctrl/pinmux.h>

struct foo_group {
        const char *name;
        const unsigned int *pins;
        const unsigned num_pins;
};

static const unsigned spi0_0_pins[] = { 0, 8, 16, 24 };
static const unsigned spi0_1_pins[] = { 38, 46, 54, 62 };
static const unsigned i2c0_pins[] = { 24, 25 };
static const unsigned mmc0_1_pins[] = { 56, 57 };
static const unsigned mmc0_2_pins[] = { 58, 59 };
static const unsigned mmc0_3_pins[] = { 60, 61, 62, 63 };

static const struct foo_group foo_groups[] = {
        {
                .name = "spi0_0_grp",
                .pins = spi0_0_pins,
                .num_pins = ARRAY_SIZE(spi0_0_pins),
        },
        {
                .name = "spi0_1_grp",
                .pins = spi0_1_pins,
                .num_pins = ARRAY_SIZE(spi0_1_pins),
        },
        {
                .name = "i2c0_grp",
                .pins = i2c0_pins,
                .num_pins = ARRAY_SIZE(i2c0_pins),
        },
        {
                .name = "mmc0_1_grp",
                .pins = mmc0_1_pins,
                .num_pins = ARRAY_SIZE(mmc0_1_pins),
        },
        {
```

(continues on next page)

(continued from previous page)

```
                .name = "mmc0_2_grp",
                .pins = mmc0_2_pins,
                .num_pins = ARRAY_SIZE(mmc0_2_pins),
        },
        {
                .name = "mmc0_3_grp",
                .pins = mmc0_3_pins,
                .num_pins = ARRAY_SIZE(mmc0_3_pins),
        },
};


static int foo_get_groups_count(struct pinctrl_dev *pctldev)
{
        return ARRAY_SIZE(foo_groups);
}

static const char *foo_get_group_name(struct pinctrl_dev *pctldev,
                                      unsigned selector)
{
        return foo_groups[selector].name;
}

static int foo_get_group_pins(struct pinctrl_dev *pctldev, unsigned␣
↪selector,
                              const unsigned ** pins,
                              unsigned * num_pins)
{
        *pins = (unsigned *) foo_groups[selector].pins;
        *num_pins = foo_groups[selector].num_pins;
        return 0;
}

static struct pinctrl_ops foo_pctrl_ops = {
        .get_groups_count = foo_get_groups_count,
        .get_group_name = foo_get_group_name,
        .get_group_pins = foo_get_group_pins,
};

struct foo_pmx_func {
        const char *name;
        const char * const *groups;
        const unsigned num_groups;
};

static const char * const spi0_groups[] = { "spi0_0_grp", "spi0_1_grp" };
static const char * const i2c0_groups[] = { "i2c0_grp" };
static const char * const mmc0_groups[] = { "mmc0_1_grp", "mmc0_2_grp",
                                            "mmc0_3_grp" };

static const struct foo_pmx_func foo_functions[] = {
        {
                .name = "spi0",
                .groups = spi0_groups,
                .num_groups = ARRAY_SIZE(spi0_groups),
        },
```

(continues on next page)

(continued from previous page)

```
        {
                .name = "i2c0",
                .groups = i2c0_groups,
                .num_groups = ARRAY_SIZE(i2c0_groups),
        },
        {
                .name = "mmc0",
                .groups = mmc0_groups,
                .num_groups = ARRAY_SIZE(mmc0_groups),
        },
};

static int foo_get_functions_count(struct pinctrl_dev *pctldev)
{
        return ARRAY_SIZE(foo_functions);
}

static const char *foo_get_fname(struct pinctrl_dev *pctldev, unsigned␣
↪selector)
{
        return foo_functions[selector].name;
}

static int foo_get_groups(struct pinctrl_dev *pctldev, unsigned selector,
                          const char * const **groups,
                          unsigned * const num_groups)
{
        *groups = foo_functions[selector].groups;
        *num_groups = foo_functions[selector].num_groups;
        return 0;
}

static int foo_set_mux(struct pinctrl_dev *pctldev, unsigned selector,
                unsigned group)
{
        u8 regbit = (1 << selector + group);

        writeb((readb(MUX)|regbit), MUX);
        return 0;
}

static struct pinmux_ops foo_pmxops = {
        .get_functions_count = foo_get_functions_count,
        .get_function_name = foo_get_fname,
        .get_function_groups = foo_get_groups,
        .set_mux = foo_set_mux,
        .strict = true,
};

/* Pinmux operations are handled by some pin controller */
static struct pinctrl_desc foo_desc = {
        ...
        .pctlops = &foo_pctrl_ops,
        .pmxops = &foo_pmxops,
};
```

In the example activating muxing 0 and 1 at the same time setting bits 0 and 1, uses one pin in common so they would collide.

The beauty of the pinmux subsystem is that since it keeps track of all pins and who is using them, it will already have denied an impossible request like that, so the driver does not need to worry about such things - when it gets a selector passed in, the pinmux subsystem makes sure no other device or GPIO assignment is already using the selected pins. Thus bits 0 and 1 in the control register will never be set at the same time.

All the above functions are mandatory to implement for a pinmux driver.

50.9 Pin control interaction with the GPIO subsystem

Note that the following implies that the use case is to use a certain pin from the Linux kernel using the API in <linux/gpio.h> with gpio_request() and similar functions. There are cases where you may be using something that your datasheet calls "GPIO mode", but actually is just an electrical configuration for a certain device. See the section below named "GPIO mode pitfalls" for more details on this scenario.

The public pinmux API contains two functions named pinctrl_gpio_request() and pinctrl_gpio_free(). These two functions shall ONLY be called from gpiolib-based drivers as part of their gpio_request() and gpio_free() semantics. Likewise the pinctrl_gpio_direction_[input|output] shall only be called from within respective gpio_direction_[input|output] gpiolib implementation.

NOTE that platforms and individual drivers shall NOT request GPIO pins to be controlled e.g. muxed in. Instead, implement a proper gpiolib driver and have that driver request proper muxing and other control for its pins.

The function list could become long, especially if you can convert every individual pin into a GPIO pin independent of any other pins, and then try the approach to define every pin as a function.

In this case, the function array would become 64 entries for each GPIO setting and then the device functions.

For this reason there are two functions a pin control driver can implement to enable only GPIO on an individual pin: .gpio_request_enable() and .gpio_disable_free().

This function will pass in the affected GPIO range identified by the pin controller core, so you know which GPIO pins are being affected by the request operation.

If your driver needs to have an indication from the framework of whether the GPIO pin shall be used for input or output you can implement the .gpio_set_direction() function. As described this shall be called from the gpiolib driver and the affected GPIO range, pin offset and desired direction will be passed along to this function.

Alternatively to using these special functions, it is fully allowed to use named functions for each GPIO pin, the pinctrl_gpio_request() will attempt to obtain the function "gpioN" where "N" is the global GPIO pin number if no special GPIO-handler is registered.

50.10 GPIO mode pitfalls

Due to the naming conventions used by hardware engineers, where "GPIO"is taken to mean different things than what the kernel does, the developer may be confused by a datasheet talking about a pin being possible to set into "GPIO mode". It appears that what hardware engineers mean with "GPIO mode" is not necessarily the use case that is implied in the kernel interface <linux/gpio.h>: a pin that you grab from kernel code and then either listen for input or drive high/low to assert/deassert some external line.

Rather hardware engineers think that "GPIO mode" means that you can software-control a few electrical properties of the pin that you would not be able to control if the pin was in some other mode, such as muxed in for a device.

The GPIO portions of a pin and its relation to a certain pin controller configuration and muxing logic can be constructed in several ways. Here are two examples:

```
(A)
                  pin config
                  logic regs
                  |               +- SPI
Physical pins --- pad --- pinmux -+- I2C
                          |       +- mmc
                          |       +- GPIO
                          pin
                          multiplex
                          logic regs
```

Here some electrical properties of the pin can be configured no matter whether the pin is used for GPIO or not. If you multiplex a GPIO onto a pin, you can also drive it high/low from "GPIO" registers. Alternatively, the pin can be controlled by a certain peripheral, while still applying desired pin config properties. GPIO functionality is thus orthogonal to any other device using the pin.

In this arrangement the registers for the GPIO portions of the pin controller, or the registers for the GPIO hardware module are likely to reside in a separate memory range only intended for GPIO driving, and the register range dealing with pin config and pin multiplexing get placed into a different memory range and a separate section of the data sheet.

A flag "strict" in struct pinmux_ops is available to check and deny simultaneous access to the same pin from GPIO and pin multiplexing consumers on hardware of this type. The pinctrl driver should set this flag accordingly.

```
(B)

                  pin config
                  logic regs
                  |               +- SPI
Physical pins --- pad --- pinmux -+- I2C
                  |       |       +- mmc
                  |       |
                  GPIO    pin
                          multiplex
                          logic regs
```

In this arrangement, the GPIO functionality can always be enabled, such that e.g. a GPIO input can be used to "spy" on the SPI/I2C/MMC signal while it is pulsed out. It is likely possible to disrupt the traffic on the pin by doing wrong things on the GPIO block, as it is never really disconnected. It is possible that the GPIO, pin config and pin multiplex registers are placed into the same memory range and the same section of the data sheet, although that need not be the case.

In some pin controllers, although the physical pins are designed in the same way as (B), the GPIO function still can't be enabled at the same time as the peripheral functions. So again the "strict" flag should be set, denying simultaneous activation by GPIO and other muxed in devices.

From a kernel point of view, however, these are different aspects of the hardware and shall be put into different subsystems:

- Registers (or fields within registers) that control electrical properties of the pin such as biasing and drive strength should be exposed through the pinctrl subsystem, as "pin configuration" settings.
- Registers (or fields within registers) that control muxing of signals from various other HW blocks (e.g. I2C, MMC, or GPIO) onto pins should be exposed through the pinctrl subsystem, as mux functions.
- Registers (or fields within registers) that control GPIO functionality such as setting a GPIO's output value, reading a GPIO's input value, or setting GPIO pin direction should be exposed through the GPIO subsystem, and if they also support interrupt capabilities, through the irqchip abstraction.

Depending on the exact HW register design, some functions exposed by the GPIO subsystem may call into the pinctrl subsystem in order to co-ordinate register settings across HW modules. In particular, this may be needed for HW with separate GPIO and pin controller HW modules, where e.g. GPIO direction is determined by a register in the pin controller HW module rather than the GPIO HW module.

Electrical properties of the pin such as biasing and drive strength may be placed at some pin-specific register in all cases or as part of the GPIO register in case (B) especially. This doesn't mean that such properties necessarily pertain to what the Linux kernel calls "GPIO".

Example: a pin is usually muxed in to be used as a UART TX line. But during system sleep, we need to put this pin into "GPIO mode" and ground it.

If you make a 1-to-1 map to the GPIO subsystem for this pin, you may start to think that you need to come up with something really complex, that the pin shall be used for UART TX and GPIO at the same time, that you will grab a pin control handle and set it to a certain state to enable UART TX to be muxed in, then twist it over to GPIO mode and use gpio_direction_output() to drive it low during sleep, then mux it over to UART TX again when you wake up and maybe even gpio_request/gpio_free as part of this cycle. This all gets very complicated.

The solution is to not think that what the datasheet calls "GPIO mode" has to be handled by the <linux/gpio.h> interface. Instead view this as a certain pin config setting. Look in e.g. <linux/pinctrl/pinconf-generic.h> and you find this in the documentation:

PIN_CONFIG_OUTPUT: this will configure the pin in output, use argument 1 to indicate high level, argument 0 to indicate low level.

So it is perfectly possible to push a pin into "GPIO mode" and drive the line low as part of the usual pin control map. So for example your UART driver may look like this:

```
#include <linux/pinctrl/consumer.h>

struct pinctrl          *pinctrl;
struct pinctrl_state    *pins_default;
struct pinctrl_state    *pins_sleep;

pins_default = pinctrl_lookup_state(uap->pinctrl, PINCTRL_STATE_DEFAULT);
pins_sleep = pinctrl_lookup_state(uap->pinctrl, PINCTRL_STATE_SLEEP);

/* Normal mode */
retval = pinctrl_select_state(pinctrl, pins_default);
/* Sleep mode */
retval = pinctrl_select_state(pinctrl, pins_sleep);
```

50.10.1 And your machine configuration may look like this:

```
static unsigned long uart_default_mode[] = {
        PIN_CONF_PACKED(PIN_CONFIG_DRIVE_PUSH_PULL, 0),
};

static unsigned long uart_sleep_mode[] = {
        PIN_CONF_PACKED(PIN_CONFIG_OUTPUT, 0),
};

static struct pinctrl_map pinmap[] __initdata = {
        PIN_MAP_MUX_GROUP("uart", PINCTRL_STATE_DEFAULT, "pinctrl-foo",
                "u0_group", "u0"),
        PIN_MAP_CONFIGS_PIN("uart", PINCTRL_STATE_DEFAULT, "pinctrl-foo",
                        "UART_TX_PIN", uart_default_mode),
        PIN_MAP_MUX_GROUP("uart", PINCTRL_STATE_SLEEP, "pinctrl-foo",
                "u0_group", "gpio-mode"),
        PIN_MAP_CONFIGS_PIN("uart", PINCTRL_STATE_SLEEP, "pinctrl-foo",
                        "UART_TX_PIN", uart_sleep_mode),
};

foo_init(void) {
        pinctrl_register_mappings(pinmap, ARRAY_SIZE(pinmap));
}
```

Here the pins we want to control are in the "u0_group" and there is some function called "u0" that can be enabled on this group of pins, and then everything is UART business as usual. But there is also some function named "gpio-mode" that can be mapped onto the same pins to move them into GPIO mode.

This will give the desired effect without any bogus interaction with the GPIO subsystem. It is just an electrical configuration used by that device when going to sleep, it might imply that the pin is set into something the datasheet calls "GPIO mode", but that is not the point: it is still used by that UART device to control the pins that pertain to that very UART driver, putting them into modes needed by the UART. GPIO in the Linux kernel sense are just some 1-bit line, and is a different use case.

How the registers are poked to attain the push or pull, and output low configuration and the muxing of the "u0" or "gpio-mode" group onto these pins is a question for the driver.

Some datasheets will be more helpful and refer to the "GPIO mode" as "low power mode" rather than anything to do with GPIO. This often means the same thing electrically speaking, but in this latter case the software engineers will usually quickly identify that this is some specific muxing or configuration rather than anything related to the GPIO API.

50.11 Board/machine configuration

Boards and machines define how a certain complete running system is put together, including how GPIOs and devices are muxed, how regulators are constrained and how the clock tree looks. Of course pinmux settings are also part of this.

A pin controller configuration for a machine looks pretty much like a simple regulator configuration, so for the example array above we want to enable i2c and spi on the second function mapping:

```
#include <linux/pinctrl/machine.h>

static const struct pinctrl_map mapping[] __initconst = {
        {
                .dev_name = "foo-spi.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "spi0",
        },
        {
                .dev_name = "foo-i2c.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "i2c0",
        },
        {
                .dev_name = "foo-mmc.0",
                .name = PINCTRL_STATE_DEFAULT,
                .type = PIN_MAP_TYPE_MUX_GROUP,
                .ctrl_dev_name = "pinctrl-foo",
                .data.mux.function = "mmc0",
        },
};
```

The dev_name here matches to the unique device name that can be used to look up the device struct (just like with clockdev or regulators). The function name must match a function provided by the pinmux driver handling this pin range.

As you can see we may have several pin controllers on the system and thus we need to specify which one of them contains the functions we wish to map.

You register this pinmux mapping to the pinmux subsystem by simply:

```
ret = pinctrl_register_mappings(mapping, ARRAY_SIZE(mapping));
```

Since the above construct is pretty common there is a helper macro to make it even more compact which assumes you want to use pinctrl-foo and position 0 for mapping, for example:

```
static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_MUX_GROUP("foo-i2c.o", PINCTRL_STATE_DEFAULT,
                          "pinctrl-foo", NULL, "i2c0"),
};
```

The mapping table may also contain pin configuration entries. It's common for each pin/group to have a number of configuration entries that affect it, so the table entries for configuration reference an array of config parameters and values. An example using the convenience macros is shown below:

```
static unsigned long i2c_grp_configs[] = {
        FOO_PIN_DRIVEN,
        FOO_PIN_PULLUP,
};

static unsigned long i2c_pin_configs[] = {
        FOO_OPEN_COLLECTOR,
        FOO_SLEW_RATE_SLOW,
};

static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_MUX_GROUP("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                          "pinctrl-foo", "i2c0", "i2c0"),
        PIN_MAP_CONFIGS_GROUP("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                              "pinctrl-foo", "i2c0", i2c_grp_configs),
        PIN_MAP_CONFIGS_PIN("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                            "pinctrl-foo", "i2c0scl", i2c_pin_configs),
        PIN_MAP_CONFIGS_PIN("foo-i2c.0", PINCTRL_STATE_DEFAULT,
                            "pinctrl-foo", "i2c0sda", i2c_pin_configs),
};
```

Finally, some devices expect the mapping table to contain certain specific named states. When running on hardware that doesn't need any pin controller configuration, the mapping table must still contain those named states, in order to explicitly indicate that the states were provided and intended to be empty. Table entry macro PIN_MAP_DUMMY_STATE serves the purpose of defining a named state without causing any pin controller to be programmed:

```
static struct pinctrl_map mapping[] __initdata = {
        PIN_MAP_DUMMY_STATE("foo-i2c.0", PINCTRL_STATE_DEFAULT),
};
```

50.12 Complex mappings

As it is possible to map a function to different groups of pins an optional .group can be specified like this:

```
...
{
        .dev_name = "foo-spi.0",
        .name = "spi0-pos-A",
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "spi0",
        .group = "spi0_0_grp",
},
{
        .dev_name = "foo-spi.0",
        .name = "spi0-pos-B",
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "spi0",
        .group = "spi0_1_grp",
},
...
```

This example mapping is used to switch between two positions for spi0 at runtime, as described further below under the heading “Runtime pinmuxing”.

Further it is possible for one named state to affect the muxing of several groups of pins, say for example in the mmc0 example above, where you can additively expand the mmc0 bus from 2 to 4 to 8 pins. If we want to use all three groups for a total of 2+2+4 = 8 pins (for an 8-bit MMC bus as is the case), we define a mapping like this:

```
...
{
        .dev_name = "foo-mmc.0",
        .name = "2bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "4bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "4bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
```

(continues on next page)

(continued from previous page)

```
        .group = "mmc0_2_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_1_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_2_grp",
},
{
        .dev_name = "foo-mmc.0",
        .name = "8bit"
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "mmc0",
        .group = "mmc0_3_grp",
},
...
```

The result of grabbing this mapping from the device with something like this (see next paragraph):

```
p = devm_pinctrl_get(dev);
s = pinctrl_lookup_state(p, "8bit");
ret = pinctrl_select_state(p, s);
```

or more simply:

```
p = devm_pinctrl_get_select(dev, "8bit");
```

Will be that you activate all the three bottom records in the mapping at once. Since they share the same name, pin controller device, function and device, and since we allow multiple groups to match to a single device, they all get selected, and they all get enabled and disable simultaneously by the pinmux core.

50.13 Pin control requests from drivers

When a device driver is about to probe the device core will automatically attempt to issue pinctrl_get_select_default() on these devices. This way driver writers do not need to add any of the boilerplate code of the type found below. However when doing fine-grained state selection and not using the “default” state, you may have to do some device driver handling of the pinctrl handles and states.

So if you just want to put the pins for a certain device into the default state and be

done with it, there is nothing you need to do besides providing the proper mapping table. The device core will take care of the rest.

Generally it is discouraged to let individual drivers get and enable pin control. So if possible, handle the pin control in platform code or some other place where you have access to all the affected struct device * pointers. In some cases where a driver needs to e.g. switch between different mux mappings at runtime this is not possible.

A typical case is if a driver needs to switch bias of pins from normal operation and going to sleep, moving from the PINCTRL_STATE_DEFAULT to PINCTRL_STATE_SLEEP at runtime, re-biasing or even re-muxing pins to save current in sleep mode.

A driver may request a certain control state to be activated, usually just the default state like this:

```
#include <linux/pinctrl/consumer.h>

struct foo_state {
struct pinctrl *p;
struct pinctrl_state *s;
...
};

foo_probe()
{
        /* Allocate a state holder named "foo" etc */
        struct foo_state *foo = ...;

        foo->p = devm_pinctrl_get(&device);
        if (IS_ERR(foo->p)) {
                /* FIXME: clean up "foo" here */
                return PTR_ERR(foo->p);
        }

        foo->s = pinctrl_lookup_state(foo->p, PINCTRL_STATE_DEFAULT);
        if (IS_ERR(foo->s)) {
                /* FIXME: clean up "foo" here */
                return PTR_ERR(s);
        }

        ret = pinctrl_select_state(foo->s);
        if (ret < 0) {
                /* FIXME: clean up "foo" here */
                return ret;
        }
}
```

This get/lookup/select/put sequence can just as well be handled by bus drivers if you don't want each and every driver to handle it and you know the arrangement on your bus.

The semantics of the pinctrl APIs are:

- pinctrl_get() is called in process context to obtain a handle to all pinctrl information for a given client device. It will allocate a struct from the kernel

memory to hold the pinmux state. All mapping table parsing or similar slow operations take place within this API.

- devm_pinctrl_get() is a variant of pinctrl_get() that causes pinctrl_put() to be called automatically on the retrieved pointer when the associated device is removed. It is recommended to use this function over plain pinctrl_get().
- pinctrl_lookup_state() is called in process context to obtain a handle to a specific state for a client device. This operation may be slow, too.
- pinctrl_select_state() programs pin controller hardware according to the definition of the state as given by the mapping table. In theory, this is a fast-path operation, since it only involved blasting some register settings into hardware. However, note that some pin controllers may have their registers on a slow/IRQ-based bus, so client devices should not assume they can call pinctrl_select_state() from non-blocking contexts.
- pinctrl_put() frees all information associated with a pinctrl handle.
- devm_pinctrl_put() is a variant of pinctrl_put() that may be used to explicitly destroy a pinctrl object returned by devm_pinctrl_get(). However, use of this function will be rare, due to the automatic cleanup that will occur even without calling it.

  pinctrl_get() must be paired with a plain pinctrl_put(). pinctrl_get() may not be paired with devm_pinctrl_put(). devm_pinctrl_get() can optionally be paired with devm_pinctrl_put(). devm_pinctrl_get() may not be paired with plain pinctrl_put().

Usually the pin control core handled the get/put pair and call out to the device drivers bookkeeping operations, like checking available functions and the associated pins, whereas select_state pass on to the pin controller driver which takes care of activating and/or deactivating the mux setting by quickly poking some registers.

The pins are allocated for your device when you issue the devm_pinctrl_get() call, after this you should be able to see this in the debugfs listing of all pins.

NOTE: the pinctrl system will return -EPROBE_DEFER if it cannot find the requested pinctrl handles, for example if the pinctrl driver has not yet registered. Thus make sure that the error path in your driver gracefully cleans up and is ready to retry the probing later in the startup process.

50.14 Drivers needing both pin control and GPIOs

Again, it is discouraged to let drivers lookup and select pin control states themselves, but again sometimes this is unavoidable.

So say that your driver is fetching its resources like this:

```
#include <linux/pinctrl/consumer.h>
#include <linux/gpio.h>

struct pinctrl *pinctrl;
int gpio;
```

(continues on next page)

(continued from previous page)

```
pinctrl = devm_pinctrl_get_select_default(&dev);
gpio = devm_gpio_request(&dev, 14, "foo");
```

Here we first request a certain pin state and then request GPIO 14 to be used. If you're using the subsystems orthogonally like this, you should nominally always get your pinctrl handle and select the desired pinctrl state BEFORE requesting the GPIO. This is a semantic convention to avoid situations that can be electrically unpleasant, you will certainly want to mux in and bias pins in a certain way before the GPIO subsystems starts to deal with them.

The above can be hidden: using the device core, the pinctrl core may be setting up the config and muxing for the pins right before the device is probing, nevertheless orthogonal to the GPIO subsystem.

But there are also situations where it makes sense for the GPIO subsystem to communicate directly with the pinctrl subsystem, using the latter as a back-end. This is when the GPIO driver may call out to the functions described in the section "Pin control interaction with the GPIO subsystem" above. This only involves per-pin multiplexing, and will be completely hidden behind the gpio_*() function namespace. In this case, the driver need not interact with the pin control subsystem at all.

If a pin control driver and a GPIO driver is dealing with the same pins and the use cases involve multiplexing, you MUST implement the pin controller as a back-end for the GPIO driver like this, unless your hardware design is such that the GPIO controller can override the pin controller's multiplexing state through hardware without the need to interact with the pin control system.

50.15 System pin control hogging

Pin control map entries can be hogged by the core when the pin controller is registered. This means that the core will attempt to call pinctrl_get(), lookup_state() and select_state() on it immediately after the pin control device has been registered.

This occurs for mapping table entries where the client device name is equal to the pin controller device name, and the state name is PINCTRL_STATE_DEFAULT:

```
{
        .dev_name = "pinctrl-foo",
        .name = PINCTRL_STATE_DEFAULT,
        .type = PIN_MAP_TYPE_MUX_GROUP,
        .ctrl_dev_name = "pinctrl-foo",
        .function = "power_func",
},
```

Since it may be common to request the core to hog a few always-applicable mux settings on the primary pin controller, there is a convenience macro for this:

```
PIN_MAP_MUX_GROUP_HOG_DEFAULT("pinctrl-foo", NULL /* group */,
                              "power_func")
```

This gives the exact same result as the above construction.

50.16 Runtime pinmuxing

It is possible to mux a certain function in and out at runtime, say to move an SPI port from one set of pins to another set of pins. Say for example for spi0 in the example above, we expose two different groups of pins for the same function, but with different named in the mapping as described under "Advanced mapping" above. So that for an SPI device, we have two states named "pos-A" and "pos-B" .

This snippet first initializes a state object for both groups (in foo_probe()), then muxes the function in the pins defined by group A, and finally muxes it in on the pins defined by group B:

```
#include <linux/pinctrl/consumer.h>

struct pinctrl *p;
struct pinctrl_state *s1, *s2;

foo_probe()
{
        /* Setup */
        p = devm_pinctrl_get(&device);
        if (IS_ERR(p))
                ...

        s1 = pinctrl_lookup_state(foo->p, "pos-A");
        if (IS_ERR(s1))
                ...

        s2 = pinctrl_lookup_state(foo->p, "pos-B");
        if (IS_ERR(s2))
                ...
}

foo_switch()
{
        /* Enable on position A */
        ret = pinctrl_select_state(s1);
        if (ret < 0)
        ...

        ...

        /* Enable on position B */
        ret = pinctrl_select_state(s2);
        if (ret < 0)
        ...

        ...
}
```

The above has to be done from process context. The reservation of the pins will be done when the state is activated, so in effect one specific pin can be used by

different functions at different times on a running system.

CHAPTER

FIFTYONE

GENERAL PURPOSE INPUT/OUTPUT (GPIO)

Contents:

51.1 Introduction

51.1.1 GPIO Interfaces

The documents in this directory give detailed instructions on how to access GPIOs in drivers, and how to write a driver for a device that provides GPIOs itself.

Due to the history of GPIO interfaces in the kernel, there are two different ways to obtain and use GPIOs:

- The descriptor-based interface is the preferred way to manipulate GPIOs, and is described by all the files in this directory excepted gpio-legacy.txt.
- The legacy integer-based interface which is considered deprecated (but still usable for compatibility reasons) is documented in gpio-legacy.txt.

The remainder of this document applies to the new descriptor-based interface. gpio-legacy.txt contains the same information applied to the legacy integer-based interface.

51.1.2 What is a GPIO?

A "General Purpose Input/Output" (GPIO) is a flexible software-controlled digital signal. They are provided from many kinds of chip, and are familiar to Linux developers working with embedded and custom hardware. Each GPIO represents a bit connected to a particular pin, or "ball" on Ball Grid Array (BGA) packages. Board schematics show which external hardware connects to which GPIOs. Drivers can be written generically, so that board setup code passes such pin configuration data to drivers.

System-on-Chip (SOC) processors heavily rely on GPIOs. In some cases, every non-dedicated pin can be configured as a GPIO; and most chips have at least several dozen of them. Programmable logic devices (like FPGAs) can easily provide GPIOs; multifunction chips like power managers, and audio codecs often have a few such pins to help with pin scarcity on SOCs; and there are also "GPIO Expander" chips that connect using the I2C or SPI serial buses. Most PC southbridges have a few dozen GPIO-capable pins (with only the BIOS firmware knowing how they're used).

The exact capabilities of GPIOs vary between systems. Common options:

- Output values are writable (high=1, low=0). Some chips also have options about how that value is driven, so that for example only one value might be driven, supporting "wire-OR"and similar schemes for the other value (notably, "open drain" signaling).
- Input values are likewise readable (1, 0). Some chips support readback of pins configured as "output", which is very useful in such "wire-OR" cases (to support bidirectional signaling). GPIO controllers may have input de-glitch/debounce logic, sometimes with software controls.
- Inputs can often be used as IRQ signals, often edge triggered but sometimes level triggered. Such IRQs may be configurable as system wakeup events, to wake the system from a low power state.
- Usually a GPIO will be configurable as either input or output, as needed by different product boards; single direction ones exist too.
- Most GPIOs can be accessed while holding spinlocks, but those accessed through a serial bus normally can't. Some systems support both types.

On a given board each GPIO is used for one specific purpose like monitoring MMC/SD card insertion/removal, detecting card write-protect status, driving a LED, configuring a transceiver, bit-banging a serial bus, poking a hardware watchdog, sensing a switch, and so on.

51.1.3 Common GPIO Properties

These properties are met through all the other documents of the GPIO interface and it is useful to understand them, especially if you need to define GPIO mappings.

Active-High and Active-Low

It is natural to assume that a GPIO is "active" when its output signal is 1 ("high"), and inactive when it is 0 ("low"). However in practice the signal of a GPIO may be inverted before is reaches its destination, or a device could decide to have different conventions about what "active" means. Such decisions should be transparent to device drivers, therefore it is possible to define a GPIO as being either active-high ("1" means "active", the default) or active-low ("0" means "active") so that drivers only need to worry about the logical signal and not about what happens at the line level.

Open Drain and Open Source

Sometimes shared signals need to use "open drain" (where only the low signal level is actually driven), or "open source" (where only the high signal level is driven) signaling. That term applies to CMOS transistors; "open collector" is used for TTL. A pullup or pulldown resistor causes the high or low signal level. This is sometimes called a "wire-AND"; or more practically, from the negative logic (low=true) perspective this is a "wire-OR".

One common example of an open drain signal is a shared active-low IRQ line. Also, bidirectional data bus signals sometimes use open drain signals.

Some GPIO controllers directly support open drain and open source outputs; many don't. When you need open drain signaling but your hardware doesn't directly support it, there's a common idiom you can use to emulate it with any GPIO pin that can be used as either an input or an output:

LOW: gpiod_direction_output(gpio, 0) ⋯this drives the signal and overrides the pullup.

HIGH: gpiod_direction_input(gpio) ⋯this turns off the output, so the pullup (or some other device) controls the signal.

The same logic can be applied to emulate open source signaling, by driving the high signal and configuring the GPIO as input for low. This open drain/open source emulation can be handled transparently by the GPIO framework.

If you are "driving" the signal high but gpiod_get_value(gpio) reports a low value (after the appropriate rise time passes), you know some other component is driving the shared signal low. That's not necessarily an error. As one common example, that's how I2C clocks are stretched: a slave that needs a slower clock delays the rising edge of SCK, and the I2C master adjusts its signaling rate accordingly.

51.2 Using GPIO Lines in Linux

The Linux kernel exists to abstract and present hardware to users. GPIO lines as such are normally not user facing abstractions. The most obvious, natural and preferred way to use GPIO lines is to let kernel hardware drivers deal with them.

For examples of already existing generic drivers that will also be good examples for any other kernel drivers you want to author, refer to Subsystem drivers using GPIO

For any kind of mass produced system you want to support, such as servers, laptops, phones, tablets, routers, and any consumer or office or business goods using appropriate kernel drivers is paramount. Submit your code for inclusion in the upstream Linux kernel when you feel it is mature enough and you will get help to refine it, see ../../process/submitting-patches.

In Linux GPIO lines also have a userspace ABI.

The userspace ABI is intended for one-off deployments. Examples are prototypes, factory lines, maker community projects, workshop specimen, production tools, industrial automation, PLC-type use cases, door controllers, in short a piece of specialized equipment that is not produced by the numbers, requiring operators to have a deep knowledge of the equipment and knows about the software-hardware interface to be set up. They should not have a natural fit to any existing kernel subsystem and not be a good fit for an operating system, because of not being reusable or abstract enough, or involving a lot of non computer hardware related policy.

Applications that have a good reason to use the industrial I/O (IIO) subsystem from userspace will likely be a good fit for using GPIO lines from userspace as well.

Do not under any circumstances abuse the GPIO userspace ABI to cut corners in any product development projects. If you use it for prototyping, then do not productify the prototype: rewrite it using proper kernel drivers. Do not under any circumstances deploy any uniform products using GPIO from userspace.

The userspace ABI is a character device for each GPIO hardware unit (GPIO chip). These devices will appear on the system as `/dev/gpiochip0` thru `/dev/gpiochipN`. Examples of how to directly use the userspace ABI can be found in the kernel tree `tools/gpio` subdirectory.

For structured and managed applications, we recommend that you make use of the libgpiod library. This provides helper abstractions, command line utlities and arbitration for multiple simultaneous consumers on the same GPIO chip.

51.3 GPIO Driver Interface

This document serves as a guide for writers of GPIO chip drivers.

Each GPIO controller driver needs to include the following header, which defines the structures used to define a GPIO driver:

```
#include <linux/gpio/driver.h>
```

51.3.1 Internal Representation of GPIOs

A GPIO chip handles one or more GPIO lines. To be considered a GPIO chip, the lines must conform to the definition: General Purpose Input/Output. If the line is not general purpose, it is not GPIO and should not be handled by a GPIO chip. The use case is the indicative: certain lines in a system may be called GPIO but serve a very particular purpose thus not meeting the criteria of a general purpose I/O. On the other hand a LED driver line may be used as a GPIO and should therefore still be handled by a GPIO chip driver.

Inside a GPIO driver, individual GPIO lines are identified by their hardware number, sometime also referred to as `offset`, which is a unique number between 0 and n-1, n being the number of GPIOs managed by the chip.

The hardware GPIO number should be something intuitive to the hardware, for example if a system uses a memory-mapped set of I/O-registers where 32 GPIO lines are handled by one bit per line in a 32-bit register, it makes sense to use hardware offsets 0..31 for these, corresponding to bits 0..31 in the register.

This number is purely internal: the hardware number of a particular GPIO line is never made visible outside of the driver.

On top of this internal number, each GPIO line also needs to have a global number in the integer GPIO namespace so that it can be used with the legacy GPIO interface. Each chip must thus have a “base” number (which can be automatically assigned), and for each GPIO line the global number will be (base + hardware number). Although the integer representation is considered deprecated, it still has many users and thus needs to be maintained.

So for example one platform could use global numbers 32-159 for GPIOs, with a controller defining 128 GPIOs at a "base" of 32; while another platform uses global numbers 0..63 with one set of GPIO controllers, 64-79 with another type of GPIO controller, and on one particular board 80-95 with an FPGA. The legacy numbers need not be contiguous; either of those platforms could also use numbers 2000-2063 to identify GPIO lines in a bank of I2C GPIO expanders.

51.3.2 Controller Drivers: gpio_chip

In the gpiolib framework each GPIO controller is packaged as a "struct gpio_chip" (see <linux/gpio/driver.h> for its complete definition) with members common to each controller of that type, these should be assigned by the driver code:

- methods to establish GPIO line direction
- methods used to access GPIO line values
- method to set electrical configuration for a given GPIO line
- method to return the IRQ number associated to a given GPIO line
- flag saying whether calls to its methods may sleep
- optional line names array to identify lines
- optional debugfs dump method (showing extra state information)
- optional base number (will be automatically assigned if omitted)
- optional label for diagnostics and GPIO chip mapping using platform data

The code implementing a gpio_chip should support multiple instances of the controller, preferably using the driver model. That code will configure each gpio_chip and issue gpiochip_add(), `gpiochip_add_data()`, or `devm_gpiochip_add_data()`. Removing a GPIO controller should be rare; use `gpiochip_remove()` when it is unavoidable.

Often a gpio_chip is part of an instance-specific structure with states not exposed by the GPIO interfaces, such as addressing, power management, and more. Chips such as audio codecs will have complex non-GPIO states.

Any debugfs dump method should normally ignore lines which haven't been requested. They can use `gpiochip_is_requested()`, which returns either NULL or the label associated with that GPIO line when it was requested.

Realtime considerations: the GPIO driver should not use spinlock_t or any sleepable APIs (like PM runtime) in its gpio_chip implementation (.get/.set and direction control callbacks) if it is expected to call GPIO APIs from atomic context on realtime kernels (inside hard IRQ handlers and similar contexts). Normally this should not be required.

GPIO electrical configuration

GPIO lines can be configured for several electrical modes of operation by using the .set_config() callback. Currently this API supports setting:

- Debouncing
- Single-ended modes (open drain/open source)
- Pull up and pull down resistor enablement

These settings are described below.

The .set_config() callback uses the same enumerators and configuration semantics as the generic pin control drivers. This is not a coincidence: it is possible to assign the .set_config() to the function `gpiochip_generic_config()` which will result in pinctrl_gpio_set_config() being called and eventually ending up in the pin control back-end "behind" the GPIO controller, usually closer to the actual pins. This way the pin controller can manage the below listed GPIO configurations.

If a pin controller back-end is used, the GPIO controller or hardware description needs to provide "GPIO ranges" mapping the GPIO line offsets to pin numbers on the pin controller so they can properly cross-reference each other.

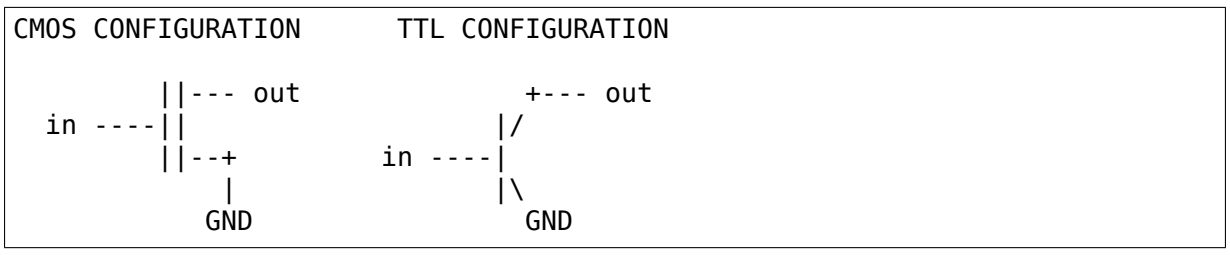
GPIO lines with debounce support

Debouncing is a configuration set to a pin indicating that it is connected to a mechanical switch or button, or similar that may bounce. Bouncing means the line is pulled high/low quickly at very short intervals for mechanical reasons. This can result in the value being unstable or irqs fireing repeatedly unless the line is debounced.

Debouncing in practice involves setting up a timer when something happens on the line, wait a little while and then sample the line again, so see if it still has the same value (low or high). This could also be repeated by a clever state machine, waiting for a line to become stable. In either case, it sets a certain number of milliseconds for debouncing, or just "on/off" if that time is not configurable.

GPIO lines with open drain/source support

Open drain (CMOS) or open collector (TTL) means the line is not actively driven high: instead you provide the drain/collector as output, so when the transistor is not open, it will present a high-impedance (tristate) to the external rail:



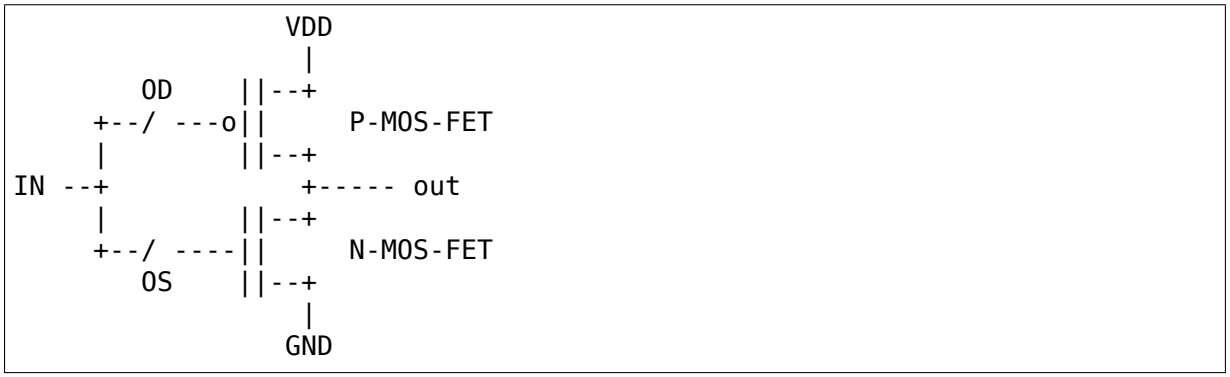
This configuration is normally used as a way to achieve one of two things:

- Level-shifting: to reach a logical level higher than that of the silicon where the output resides.
- Inverse wire-OR on an I/O line, for example a GPIO line, making it possible for any driving stage on the line to drive it low even if any other output to the same line is simultaneously driving it high. A special case of this is driving the SCL and SDA lines of an I2C bus, which is by definition a wire-OR bus.

Both use cases require that the line be equipped with a pull-up resistor. This resistor will make the line tend to high level unless one of the transistors on the rail actively pulls it down.

The level on the line will go as high as the VDD on the pull-up resistor, which may be higher than the level supported by the transistor, achieving a level-shift to the higher VDD.

Integrated electronics often have an output driver stage in the form of a CMOS "totem-pole" with one N-MOS and one P-MOS transistor where one of them drives the line high and one of them drives the line low. This is called a push-pull output. The "totem-pole" looks like so:



The desired output signal (e.g. coming directly from some GPIO output register) arrives at IN. The switches named "OD" and "OS" are normally closed, creating a push-pull circuit.

Consider the little "switches" named "OD" and "OS" that enable/disable the P-MOS or N-MOS transistor right after the split of the input. As you can see, either transistor will go totally numb if this switch is open. The totem-pole is then halved and give high impedance instead of actively driving the line high or low respectively. That is usually how software-controlled open drain/source works.

Some GPIO hardware come in open drain / open source configuration. Some are hard-wired lines that will only support open drain or open source no matter what: there is only one transistor there. Some are software-configurable: by flipping a bit in a register the output can be configured as open drain or open source, in practice by flicking open the switches labeled "OD" and "OS" in the drawing above.

By disabling the P-MOS transistor, the output can be driven between GND and high impedance (open drain), and by disabling the N-MOS transistor, the output can be driven between VDD and high impedance (open source). In the first case, a pull-up resistor is needed on the outgoing rail to complete the circuit, and in the second case, a pull-down resistor is needed on the rail.

Hardware that supports open drain or open source or both, can implement a spe-

cial callback in the gpio_chip: .set_config() that takes a generic pinconf packed value telling whether to configure the line as open drain, open source or push-pull. This will happen in response to the GPIO_OPEN_DRAIN or GPIO_OPEN_SOURCE flag set in the machine file, or coming from other hardware descriptions.

If this state can not be configured in hardware, i.e. if the GPIO hardware does not support open drain/open source in hardware, the GPIO library will instead use a trick: when a line is set as output, if the line is flagged as open drain, and the IN output value is low, it will be driven low as usual. But if the IN output value is set to high, it will instead NOT be driven high, instead it will be switched to input, as input mode is high impedance, thus achieveing an "open drain emulation" of sorts: electrically the behaviour will be identical, with the exception of possible hardware glitches when switching the mode of the line.

For open source configuration the same principle is used, just that instead of actively driving the line low, it is set to input.

GPIO lines with pull up/down resistor support

A GPIO line can support pull-up/down using the .set_config() callback. This means that a pull up or pull-down resistor is available on the output of the GPIO line, and this resistor is software controlled.

In discrete designs, a pull-up or pull-down resistor is simply soldered on the circuit board. This is not something we deal with or model in software. The most you will think about these lines is that they will very likely be configured as open drain or open source (see the section above).

The .set_config() callback can only turn pull up or down on and off, and will no have any semantic knowledge about the resistance used. It will only say switch a bit in a register enabling or disabling pull-up or pull-down.

If the GPIO line supports shunting in different resistance values for the pull-up or pull-down resistor, the GPIO chip callback .set_config() will not suffice. For these complex use cases, a combined GPIO chip and pin controller need to be implemented, as the pin config interface of a pin controller supports more versatile control over electrical properties and can handle different pull-up or pull-down resistance values.

51.3.3 GPIO drivers providing IRQs

It is custom that GPIO drivers (GPIO chips) are also providing interrupts, most often cascaded off a parent interrupt controller, and in some special cases the GPIO logic is melded with a SoC's primary interrupt controller.

The IRQ portions of the GPIO block are implemented using an irq_chip, using the header <linux/irq.h>. So this combined driver is utilizing two sub- systems simultaneously: gpio and irq.

It is legal for any IRQ consumer to request an IRQ from any irqchip even if it is a combined GPIO+IRQ driver. The basic premise is that gpio_chip and irq_chip are orthogonal, and offering their services independent of each other.

`gpiod_to_irq()` is just a convenience function to figure out the IRQ for a certain GPIO line and should not be relied upon to have been called before the IRQ is used.

Always prepare the hardware and make it ready for action in respective callbacks from the GPIO and irq_chip APIs. Do not rely on `gpiod_to_irq()` having been called first.

We can divide GPIO irqchips in two broad categories:

- CASCADED INTERRUPT CHIPS: this means that the GPIO chip has one common interrupt output line, which is triggered by any enabled GPIO line on that chip. The interrupt output line will then be routed to an parent interrupt controller one level up, in the most simple case the systems primary interrupt controller. This is modeled by an irqchip that will inspect bits inside the GPIO controller to figure out which line fired it. The irqchip part of the driver needs to inspect registers to figure this out and it will likely also need to acknowledge that it is handling the interrupt by clearing some bit (sometime implicitly, by just reading a status register) and it will often need to set up the configuration such as edge sensitivity (rising or falling edge, or high/low level interrupt for example).
- HIERARCHICAL INTERRUPT CHIPS: this means that each GPIO line has a dedicated irq line to a parent interrupt controller one level up. There is no need to inquire the GPIO hardware to figure out which line has fired, but it may still be necessary to acknowledge the interrupt and set up configuration such as edge sensitivity.

Realtime considerations: a realtime compliant GPIO driver should not use spinlock_t or any sleepable APIs (like PM runtime) as part of its irqchip implementation.

- spinlock_t should be replaced with raw_spinlock_t.[1]
- If sleepable APIs have to be used, these can be done from the .irq_bus_lock() and .irq_bus_unlock() callbacks, as these are the only slowpath callbacks on an irqchip. Create the callbacks if needed.[2]

Cascaded GPIO irqchips

Cascaded GPIO irqchips usually fall in one of three categories:

- CHAINED CASCADED GPIO IRQCHIPS: these are usually the type that is embedded on an SoC. This means that there is a fast IRQ flow handler for the GPIOs that gets called in a chain from the parent IRQ handler, most typically the system interrupt controller. This means that the GPIO irqchip handler will be called immediately from the parent irqchip, while holding the IRQs disabled. The GPIO irqchip will then end up calling something like this sequence in its interrupt handler:

```
static irqreturn_t foo_gpio_irq(int irq, void *data)
    chained_irq_enter(...);
    generic_handle_irq(...);
    chained_irq_exit(...);
```

Chained GPIO irqchips typically can NOT set the .can_sleep flag on struct gpio_chip, as everything happens directly in the callbacks: no slow bus traffic like I2C can be used.

Realtime considerations: Note that chained IRQ handlers will not be forced threaded on -RT. As a result, spinlock_t or any sleepable APIs (like PM runtime) can't be used in a chained IRQ handler.

If required (and if it can't be converted to the nested threaded GPIO irqchip, see below) a chained IRQ handler can be converted to generic irq handler and this way it will become a threaded IRQ handler on -RT and a hard IRQ handler on non-RT (for example, see [3]).

The generic_handle_irq() is expected to be called with IRQ disabled, so the IRQ core will complain if it is called from an IRQ handler which is forced to a thread. The "fake?" raw lock can be used to work around this problem:

```
raw_spinlock_t wa_lock;
static irqreturn_t omap_gpio_irq_handler(int irq, void *gpiobank)
        unsigned long wa_lock_flags;
        raw_spin_lock_irqsave(&bank->wa_lock, wa_lock_flags);
        generic_handle_irq(irq_find_mapping(bank->chip.irq.domain,␣
↪bit));
        raw_spin_unlock_irqrestore(&bank->wa_lock, wa_lock_flags);
```

- GENERIC CHAINED GPIO IRQCHIPS: these are the same as "CHAINED GPIO irqchips", but chained IRQ handlers are not used. Instead GPIO IRQs dispatching is performed by generic IRQ handler which is configured using request_irq(). The GPIO irqchip will then end up calling something like this sequence in its interrupt handler:

```
static irqreturn_t gpio_rcar_irq_handler(int irq, void *dev_id)
    for each detected GPIO IRQ
        generic_handle_irq(...);
```

Realtime considerations: this kind of handlers will be forced threaded on -RT, and as result the IRQ core will complain that generic_handle_irq() is called with IRQ enabled and the same work-around as for "CHAINED GPIO irqchips" can be applied.

- NESTED THREADED GPIO IRQCHIPS: these are off-chip GPIO expanders and any other GPIO irqchip residing on the other side of a sleeping bus such as I2C or SPI.

  Of course such drivers that need slow bus traffic to read out IRQ status and similar, traffic which may in turn incur other IRQs to happen, cannot be handled in a quick IRQ handler with IRQs disabled. Instead they need to spawn a thread and then mask the parent IRQ line until the interrupt is handled by the driver. The hallmark of this driver is to call something like this in its interrupt handler:

```
static irqreturn_t foo_gpio_irq(int irq, void *data)
    ...
    handle_nested_irq(irq);
```

The hallmark of threaded GPIO irqchips is that they set the .can_sleep flag on

struct gpio_chip to true, indicating that this chip may sleep when accessing the GPIOs.

These kinds of irqchips are inherently realtime tolerant as they are already set up to handle sleeping contexts.

Infrastructure helpers for GPIO irqchips

To help out in handling the set-up and management of GPIO irqchips and the associated irqdomain and resource allocation callbacks. These are activated by selecting the Kconfig symbol GPIOLIB_IRQCHIP. If the symbol IRQ_DOMAIN_HIERARCHY is also selected, hierarchical helpers will also be provided. A big portion of overhead code will be managed by gpiolib, under the assumption that your interrupts are 1-to-1-mapped to the GPIO line index:

GPIO line offset	Hardware IRQ
0	0
1	1
2	2
…	…
ngpio-1	ngpio-1

If some GPIO lines do not have corresponding IRQs, the bitmask valid_mask and the flag need_valid_mask in gpio_irq_chip can be used to mask off some lines as invalid for associating with IRQs.

The preferred way to set up the helpers is to fill in the struct gpio_irq_chip inside struct gpio_chip before adding the gpio_chip. If you do this, the additional irq_chip will be set up by gpiolib at the same time as setting up the rest of the GPIO functionality. The following is a typical example of a cascaded interrupt handler using gpio_irq_chip:

```
/* Typical state container with dynamic irqchip */
struct my_gpio {
    struct gpio_chip gc;
    struct irq_chip irq;
};

int irq; /* from platform etc */
struct my_gpio *g;
struct gpio_irq_chip *girq;

/* Set up the irqchip dynamically */
g->irq.name = "my_gpio_irq";
g->irq.irq_ack = my_gpio_ack_irq;
g->irq.irq_mask = my_gpio_mask_irq;
g->irq.irq_unmask = my_gpio_unmask_irq;
g->irq.irq_set_type = my_gpio_set_irq_type;

/* Get a pointer to the gpio_irq_chip */
girq = &g->gc.irq;
girq->chip = &g->irq;
girq->parent_handler = ftgpio_gpio_irq_handler;
```

(continues on next page)

(continued from previous page)

```
girq->num_parents = 1;
girq->parents = devm_kcalloc(dev, 1, sizeof(*girq->parents),
                             GFP_KERNEL);
if (!girq->parents)
    return -ENOMEM;
girq->default_type = IRQ_TYPE_NONE;
girq->handler = handle_bad_irq;
girq->parents[0] = irq;

return devm_gpiochip_add_data(dev, &g->gc, g);
```

The helper support using hierarchical interrupt controllers as well. In this case the typical set-up will look like this:

```
/* Typical state container with dynamic irqchip */
struct my_gpio {
    struct gpio_chip gc;
    struct irq_chip irq;
    struct fwnode_handle *fwnode;
};

int irq; /* from platform etc */
struct my_gpio *g;
struct gpio_irq_chip *girq;

/* Set up the irqchip dynamically */
g->irq.name = "my_gpio_irq";
g->irq.irq_ack = my_gpio_ack_irq;
g->irq.irq_mask = my_gpio_mask_irq;
g->irq.irq_unmask = my_gpio_unmask_irq;
g->irq.irq_set_type = my_gpio_set_irq_type;

/* Get a pointer to the gpio_irq_chip */
girq = &g->gc.irq;
girq->chip = &g->irq;
girq->default_type = IRQ_TYPE_NONE;
girq->handler = handle_bad_irq;
girq->fwnode = g->fwnode;
girq->parent_domain = parent;
girq->child_to_parent_hwirq = my_gpio_child_to_parent_hwirq;

return devm_gpiochip_add_data(dev, &g->gc, g);
```

As you can see pretty similar, but you do not supply a parent handler for the IRQ, instead a parent irqdomain, an fwnode for the hardware and a funcion .child_to_parent_hwirq() that has the purpose of looking up the parent hardware irq from a child (i.e. this gpio chip) hardware irq. As always it is good to look at examples in the kernel tree for advice on how to find the required pieces.

The old way of adding irqchips to gpiochips after registration is also still available but we try to move away from this:

- DEPRECATED: gpiochip_irqchip_add(): adds a chained cascaded irqchip to a gpiochip. It will pass the struct gpio_chip* for the chip to all IRQ callbacks, so the callbacks need to embed the gpio_chip in its state container and obtain a pointer to the container using `container_of()`. (See Documentation/driver-

api/driver-model/design-patterns.rst)

- gpiochip_irqchip_add_nested(): adds a nested cascaded irqchip to a gpiochip, as discussed above regarding different types of cascaded irqchips. The cascaded irq has to be handled by a threaded interrupt handler. Apart from that it works exactly like the chained irqchip.
- `gpiochip_set_nested_irqchip()`: sets up a nested cascaded irq handler for a gpio_chip from a parent IRQ. As the parent IRQ has usually been explicitly requested by the driver, this does very little more than mark all the child IRQs as having the other IRQ as parent.

If there is a need to exclude certain GPIO lines from the IRQ domain handled by these helpers, we can set .irq.need_valid_mask of the gpiochip before `devm_gpiochip_add_data()` or `gpiochip_add_data()` is called. This allocates an .irq.valid_mask with as many bits set as there are GPIO lines in the chip, each bit representing line 0..n-1. Drivers can exclude GPIO lines by clearing bits from this mask. The mask must be filled in before gpiochip_irqchip_add() or gpiochip_irqchip_add_nested() is called.

To use the helpers please keep the following in mind:

- Make sure to assign all relevant members of the struct gpio_chip so that the irqchip can initialize. E.g. .dev and .can_sleep shall be set up properly.
- Nominally set all handlers to handle_bad_irq() in the setup call and pass handle_bad_irq() as flow handler parameter in gpiochip_irqchip_add() if it is expected for GPIO driver that irqchip .set_type() callback will be called before using/enabling each GPIO IRQ. Then set the handler to handle_level_irq() and/or handle_edge_irq() in the irqchip .set_type() callback depending on what your controller supports and what is requested by the consumer.

Locking IRQ usage

Since GPIO and irq_chip are orthogonal, we can get conflicts between different use cases. For example a GPIO line used for IRQs should be an input line, it does not make sense to fire interrupts on an output GPIO.

If there is competition inside the subsystem which side is using the resource (a certain GPIO line and register for example) it needs to deny certain operations and keep track of usage inside of the gpiolib subsystem.

Input GPIOs can be used as IRQ signals. When this happens, a driver is requested to mark the GPIO as being used as an IRQ:

```
int gpiochip_lock_as_irq(struct gpio_chip *chip, unsigned int offset)
```

This will prevent the use of non-irq related GPIO APIs until the GPIO IRQ lock is released:

```
void gpiochip_unlock_as_irq(struct gpio_chip *chip, unsigned int offset)
```

When implementing an irqchip inside a GPIO driver, these two functions should typically be called in the .startup() and .shutdown() callbacks from the irqchip.

When using the gpiolib irqchip helpers, these callbacks are automatically assigned.

Disabling and enabling IRQs

In some (fringe) use cases, a driver may be using a GPIO line as input for IRQs, but occasionally switch that line over to drive output and then back to being an input with interrupts again. This happens on things like CEC (Consumer Electronics Control).

When a GPIO is used as an IRQ signal, then gpiolib also needs to know if the IRQ is enabled or disabled. In order to inform gpiolib about this, the irqchip driver should call:

```
void gpiochip_disable_irq(struct gpio_chip *chip, unsigned int offset)
```

This allows drivers to drive the GPIO as an output while the IRQ is disabled. When the IRQ is enabled again, a driver should call:

```
void gpiochip_enable_irq(struct gpio_chip *chip, unsigned int offset)
```

When implementing an irqchip inside a GPIO driver, these two functions should typically be called in the .irq_disable() and .irq_enable() callbacks from the irqchip.

When using the gpiolib irqchip helpers, these callbacks are automatically assigned.

Real-Time compliance for GPIO IRQ chips

Any provider of irqchips needs to be carefully tailored to support Real-Time preemption. It is desirable that all irqchips in the GPIO subsystem keep this in mind and do the proper testing to assure they are real time-enabled.

So, pay attention on above realtime considerations in the documentation.

The following is a checklist to follow when preparing a driver for real-time compliance:

- ensure spinlock_t is not used as part irq_chip implementation
- ensure that sleepable APIs are not used as part irq_chip implementation If sleepable APIs have to be used, these can be done from the .irq_bus_lock() and .irq_bus_unlock() callbacks
- Chained GPIO irqchips: ensure spinlock_t or any sleepable APIs are not used from the chained IRQ handler
- Generic chained GPIO irqchips: take care about generic_handle_irq() calls and apply corresponding work-around
- Chained GPIO irqchips: get rid of the chained IRQ handler and use generic irq handler if possible
- regmap_mmio: it is possible to disable internal locking in regmap by setting .disable_locking and handling the locking in the GPIO driver

- Test your driver with the appropriate in-kernel real-time test cases for both level and edge IRQs
- [1] http://www.spinics.net/lists/linux-omap/msg120425.html
- [2] https://lkml.org/lkml/2015/9/25/494
- [3] https://lkml.org/lkml/2015/9/25/495

51.3.4 Requesting self-owned GPIO pins

Sometimes it is useful to allow a GPIO chip driver to request its own GPIO descriptors through the gpiolib API. A GPIO driver can use the following functions to request and free descriptors:

```
struct gpio_desc *gpiochip_request_own_desc(struct gpio_desc *desc,
                                            u16 hwnum,
                                            const char *label,
                                            enum gpiod_flags flags)

void gpiochip_free_own_desc(struct gpio_desc *desc)
```

Descriptors requested with `gpiochip_request_own_desc()` must be released with `gpiochip_free_own_desc()`.

These functions must be used with care since they do not affect module use count. Do not use the functions to request gpio descriptors not owned by the calling driver.

51.4 GPIO Descriptor Consumer Interface

This document describes the consumer interface of the GPIO framework. Note that it describes the new descriptor-based interface. For a description of the deprecated integer-based GPIO interface please refer to gpio-legacy.txt.

51.4.1 Guidelines for GPIOs consumers

Drivers that can't work without standard GPIO calls should have Kconfig entries that depend on GPIOLIB or select GPIOLIB. The functions that allow a driver to obtain and use GPIOs are available by including the following file:

```
#include <linux/gpio/consumer.h>
```

There are static inline stubs for all functions in the header file in the case where GPIOLIB is disabled. When these stubs are called they will emit warnings. These stubs are used for two use cases:

- Simple compile coverage with e.g. COMPILE_TEST - it does not matter that the current platform does not enable or select GPIOLIB because we are not going to execute the system anyway.
- Truly optional GPIOLIB support - where the driver does not really make use of the GPIOs on certain compile-time configurations for certain systems, but

will use it under other compile-time configurations. In this case the consumer must make sure not to call into these functions, or the user will be met with console warnings that may be perceived as intimidating.

All the functions that work with the descriptor-based GPIO interface are prefixed with `gpiod_`. The `gpio_` prefix is used for the legacy interface. No other function in the kernel should use these prefixes. The use of the legacy functions is strongly discouraged, new code should use <linux/gpio/consumer.h> and descriptors exclusively.

51.4.2 Obtaining and Disposing GPIOs

With the descriptor-based interface, GPIOs are identified with an opaque, non-forgeable handler that must be obtained through a call to one of the `gpiod_get()` functions. Like many other kernel subsystems, `gpiod_get()` takes the device that will use the GPIO and the function the requested GPIO is supposed to fulfill:

```
struct gpio_desc *gpiod_get(struct device *dev, const char *con_id,
                            enum gpiod_flags flags)
```

If a function is implemented by using several GPIOs together (e.g. a simple LED device that displays digits), an additional index argument can be specified:

```
struct gpio_desc *gpiod_get_index(struct device *dev,
                                  const char *con_id, unsigned int idx,
                                  enum gpiod_flags flags)
```

For a more detailed description of the con_id parameter in the DeviceTree case see Documentation/driver-api/gpio/board.rst

The flags parameter is used to optionally specify a direction and initial value for the GPIO. Values can be:

- GPIOD_ASIS or 0 to not initialize the GPIO at all. The direction must be set later with one of the dedicated functions.
- GPIOD_IN to initialize the GPIO as input.
- GPIOD_OUT_LOW to initialize the GPIO as output with a value of 0.
- GPIOD_OUT_HIGH to initialize the GPIO as output with a value of 1.
- GPIOD_OUT_LOW_OPEN_DRAIN same as GPIOD_OUT_LOW but also enforce the line to be electrically used with open drain.
- GPIOD_OUT_HIGH_OPEN_DRAIN same as GPIOD_OUT_HIGH but also enforce the line to be electrically used with open drain.

The two last flags are used for use cases where open drain is mandatory, such as I2C: if the line is not already configured as open drain in the mappings (see board.txt), then open drain will be enforced anyway and a warning will be printed that the board configuration needs to be updated to match the use case.

Both functions return either a valid GPIO descriptor, or an error code checkable with IS_ERR() (they will never return a NULL pointer). -ENOENT will be returned if and only if no GPIO has been assigned to the device/function/index triplet, other error codes are used for cases where a GPIO has been assigned but

an error occurred while trying to acquire it. This is useful to discriminate between mere errors and an absence of GPIO for optional GPIO parameters. For the common pattern where a GPIO is optional, the `gpiod_get_optional()` and `gpiod_get_index_optional()` functions can be used. These functions return NULL instead of -ENOENT if no GPIO has been assigned to the requested function:

```
struct gpio_desc *gpiod_get_optional(struct device *dev,
                                     const char *con_id,
                                     enum gpiod_flags flags)

struct gpio_desc *gpiod_get_index_optional(struct device *dev,
                                           const char *con_id,
                                           unsigned int index,
                                           enum gpiod_flags flags)
```

Note that gpio_get*_optional() functions (and their managed variants), unlike the rest of gpiolib API, also return NULL when gpiolib support is disabled. This is helpful to driver authors, since they do not need to special case -ENOSYS return codes. System integrators should however be careful to enable gpiolib on systems that need it.

For a function using multiple GPIOs all of those can be obtained with one call:

```
struct gpio_descs *gpiod_get_array(struct device *dev,
                                   const char *con_id,
                                   enum gpiod_flags flags)
```

This function returns a struct gpio_descs which contains an array of descriptors. It also contains a pointer to a gpiolib private structure which, if passed back to get/set array functions, may speed up I/O proocessing:

```
struct gpio_descs {
        struct gpio_array *info;
        unsigned int ndescs;
        struct gpio_desc *desc[];
}
```

The following function returns NULL instead of -ENOENT if no GPIOs have been assigned to the requested function:

```
struct gpio_descs *gpiod_get_array_optional(struct device *dev,
                                            const char *con_id,
                                            enum gpiod_flags flags)
```

Device-managed variants of these functions are also defined:

```
struct gpio_desc *devm_gpiod_get(struct device *dev, const char *con_id,
                                 enum gpiod_flags flags)

struct gpio_desc *devm_gpiod_get_index(struct device *dev,
                                       const char *con_id,
                                       unsigned int idx,
                                       enum gpiod_flags flags)

struct gpio_desc *devm_gpiod_get_optional(struct device *dev,
                                          const char *con_id,
```

(continues on next page)

(continued from previous page)

```
                                            enum gpiod_flags flags)

struct gpio_desc *devm_gpiod_get_index_optional(struct device *dev,
                                                const char *con_id,
                                                unsigned int index,
                                                enum gpiod_flags flags)

struct gpio_descs *devm_gpiod_get_array(struct device *dev,
                                        const char *con_id,
                                        enum gpiod_flags flags)

struct gpio_descs *devm_gpiod_get_array_optional(struct device *dev,
                                                 const char *con_id,
                                                 enum gpiod_flags flags)
```

A GPIO descriptor can be disposed of using the `gpiod_put()` function:

```
void gpiod_put(struct gpio_desc *desc)
```

For an array of GPIOs this function can be used:

```
void gpiod_put_array(struct gpio_descs *descs)
```

It is strictly forbidden to use a descriptor after calling these functions. It is also not allowed to individually release descriptors (using `gpiod_put()`) from an array acquired with `gpiod_get_array()`.

The device-managed variants are, unsurprisingly:

```
void devm_gpiod_put(struct device *dev, struct gpio_desc *desc)

void devm_gpiod_put_array(struct device *dev, struct gpio_descs *descs)
```

51.4.3 Using GPIOs

Setting Direction

The first thing a driver must do with a GPIO is setting its direction. If no direction-setting flags have been given to gpiod_get*(), this is done by invoking one of the gpiod_direction_*() functions:

```
int gpiod_direction_input(struct gpio_desc *desc)
int gpiod_direction_output(struct gpio_desc *desc, int value)
```

The return value is zero for success, else a negative errno. It should be checked, since the get/set calls don't return errors and since misconfiguration is possible. You should normally issue these calls from a task context. However, for spinlock-safe GPIOs it is OK to use them before tasking is enabled, as part of early board setup.

For output GPIOs, the value provided becomes the initial output value. This helps avoid signal glitching during system startup.

A driver can also query the current direction of a GPIO:

```
int gpiod_get_direction(const struct gpio_desc *desc)
```

This function returns 0 for output, 1 for input, or an error code in case of error.

Be aware that there is no default direction for GPIOs. Therefore, **using a GPIO without setting its direction first is illegal and will result in undefined behavior!**

Spinlock-Safe GPIO Access

Most GPIO controllers can be accessed with memory read/write instructions. Those don't need to sleep, and can safely be done from inside hard (non-threaded) IRQ handlers and similar contexts.

Use the following calls to access GPIOs from an atomic context:

```
int gpiod_get_value(const struct gpio_desc *desc);
void gpiod_set_value(struct gpio_desc *desc, int value);
```

The values are boolean, zero for low, nonzero for high. When reading the value of an output pin, the value returned should be what's seen on the pin. That won't always match the specified output value, because of issues including open-drain signaling and output latencies.

The get/set calls do not return errors because "invalid GPIO" should have been reported earlier from gpiod_direction_*(). However, note that not all platforms can read the value of output pins; those that can't should always return zero. Also, using these calls for GPIOs that can't safely be accessed without sleeping (see below) is an error.

GPIO Access That May Sleep

Some GPIO controllers must be accessed using message based buses like I2C or SPI. Commands to read or write those GPIO values require waiting to get to the head of a queue to transmit a command and get its response. This requires sleeping, which can't be done from inside IRQ handlers.

Platforms that support this type of GPIO distinguish them from other GPIOs by returning nonzero from this call:

```
int gpiod_cansleep(const struct gpio_desc *desc)
```

To access such GPIOs, a different set of accessors is defined:

```
int gpiod_get_value_cansleep(const struct gpio_desc *desc)
void gpiod_set_value_cansleep(struct gpio_desc *desc, int value)
```

Accessing such GPIOs requires a context which may sleep, for example a threaded IRQ handler, and those accessors must be used instead of spinlock-safe accessors without the cansleep() name suffix.

Other than the fact that these accessors might sleep, and will work on GPIOs that can't be accessed from hardIRQ handlers, these calls act the same as the spinlock-safe calls.

The active low and open drain semantics

As a consumer should not have to care about the physical line level, all of the gpiod_set_value_xxx() or gpiod_set_array_value_xxx() functions operate with the logical value. With this they take the active low property into account. This means that they check whether the GPIO is configured to be active low, and if so, they manipulate the passed value before the physical line level is driven.

The same is applicable for open drain or open source output lines: those do not actively drive their output high (open drain) or low (open source), they just switch their output to a high impedance value. The consumer should not need to care. (For details read about open drain in driver.txt.)

With this, all the gpiod_set_(array)_value_xxx() functions interpret the parameter "value" as "asserted" ("1") or "de-asserted" ("0"). The physical line level will be driven accordingly.

As an example, if the active low property for a dedicated GPIO is set, and the gpiod_set_(array)_value_xxx() passes "asserted" ("1"), the physical line level will be driven low.

To summarize:

```
Function (example)                 line property          physical line
gpiod_set_raw_value(desc, 0);      don't care             low
gpiod_set_raw_value(desc, 1);      don't care             high
gpiod_set_value(desc, 0);          default (active high)  low
gpiod_set_value(desc, 1);          default (active high)  high
gpiod_set_value(desc, 0);          active low             high
gpiod_set_value(desc, 1);          active low             low
gpiod_set_value(desc, 0);          open drain             low
gpiod_set_value(desc, 1);          open drain             high impedance
gpiod_set_value(desc, 0);          open source            high impedance
gpiod_set_value(desc, 1);          open source            high
```

It is possible to override these semantics using the set_raw/get_raw functions but it should be avoided as much as possible, especially by system-agnostic drivers which should not need to care about the actual physical line level and worry about the logical value instead.

Accessing raw GPIO values

Consumers exist that need to manage the logical state of a GPIO line, i.e. the value their device will actually receive, no matter what lies between it and the GPIO line.

The following set of calls ignore the active-low or open drain property of a GPIO and work on the raw line value:

```
int gpiod_get_raw_value(const struct gpio_desc *desc)
void gpiod_set_raw_value(struct gpio_desc *desc, int value)
int gpiod_get_raw_value_cansleep(const struct gpio_desc *desc)
void gpiod_set_raw_value_cansleep(struct gpio_desc *desc, int value)
int gpiod_direction_output_raw(struct gpio_desc *desc, int value)
```

The active low state of a GPIO can also be queried using the following call:

```
int gpiod_is_active_low(const struct gpio_desc *desc)
```

Note that these functions should only be used with great moderation; a driver should not have to care about the physical line level or open drain semantics.

Access multiple GPIOs with a single function call

The following functions get or set the values of an array of GPIOs:

```
int gpiod_get_array_value(unsigned int array_size,
                          struct gpio_desc **desc_array,
                          struct gpio_array *array_info,
                          unsigned long *value_bitmap);
int gpiod_get_raw_array_value(unsigned int array_size,
                              struct gpio_desc **desc_array,
                              struct gpio_array *array_info,
                              unsigned long *value_bitmap);
int gpiod_get_array_value_cansleep(unsigned int array_size,
                                   struct gpio_desc **desc_array,
                                   struct gpio_array *array_info,
                                   unsigned long *value_bitmap);
int gpiod_get_raw_array_value_cansleep(unsigned int array_size,
                                   struct gpio_desc **desc_array,
                                   struct gpio_array *array_info,
                                   unsigned long *value_bitmap);

int gpiod_set_array_value(unsigned int array_size,
                          struct gpio_desc **desc_array,
                          struct gpio_array *array_info,
                          unsigned long *value_bitmap)
int gpiod_set_raw_array_value(unsigned int array_size,
                              struct gpio_desc **desc_array,
                              struct gpio_array *array_info,
                              unsigned long *value_bitmap)
int gpiod_set_array_value_cansleep(unsigned int array_size,
                                   struct gpio_desc **desc_array,
                                   struct gpio_array *array_info,
                                   unsigned long *value_bitmap)
int gpiod_set_raw_array_value_cansleep(unsigned int array_size,
                                       struct gpio_desc **desc_array,
                                       struct gpio_array *array_info,
                                       unsigned long *value_bitmap)
```

The array can be an arbitrary set of GPIOs. The functions will try to access GPIOs belonging to the same bank or chip simultaneously if supported by the corresponding chip driver. In that case a significantly improved performance can be expected. If simultaneous access is not possible the GPIOs will be accessed sequentially.

The functions take three arguments:

- array_size - the number of array elements
- desc_array - an array of GPIO descriptors
- array_info - optional information obtained from `gpiod_get_array()`

- **value_bitmap - a bitmap to store the GPIOs' values (get) or** a bitmap of values to assign to the GPIOs (set)

The descriptor array can be obtained using the `gpiod_get_array()` function or one of its variants. If the group of descriptors returned by that function matches the desired group of GPIOs, those GPIOs can be accessed by simply using the struct gpio_descs returned by `gpiod_get_array()`:

```
struct gpio_descs *my_gpio_descs = gpiod_get_array(...);
gpiod_set_array_value(my_gpio_descs->ndescs, my_gpio_descs->desc,
                      my_gpio_descs->info, my_gpio_value_bitmap);
```

It is also possible to access a completely arbitrary array of descriptors. The descriptors may be obtained using any combination of `gpiod_get()` and `gpiod_get_array()`. Afterwards the array of descriptors has to be setup manually before it can be passed to one of the above functions. In that case, array_info should be set to NULL.

Note that for optimal performance GPIOs belonging to the same chip should be contiguous within the array of descriptors.

Still better performance may be achieved if array indexes of the descriptors match hardware pin numbers of a single chip. If an array passed to a get/set array function matches the one obtained from `gpiod_get_array()` and array_info associated with the array is also passed, the function may take a fast bitmap processing path, passing the value_bitmap argument directly to the respective .get/set_multiple() callback of the chip. That allows for utilization of GPIO banks as data I/O ports without much loss of performance.

The return value of `gpiod_get_array_value()` and its variants is 0 on success or negative on error. Note the difference to `gpiod_get_value()`, which returns 0 or 1 on success to convey the GPIO value. With the array functions, the GPIO values are stored in value_array rather than passed back as return value.

GPIOs mapped to IRQs

GPIO lines can quite often be used as IRQs. You can get the IRQ number corresponding to a given GPIO using the following call:

```
int gpiod_to_irq(const struct gpio_desc *desc)
```

It will return an IRQ number, or a negative errno code if the mapping can't be done (most likely because that particular GPIO cannot be used as IRQ). It is an unchecked error to use a GPIO that wasn't set up as an input using `gpiod_direction_input()`, or to use an IRQ number that didn't originally come from `gpiod_to_irq()`. `gpiod_to_irq()` is not allowed to sleep.

Non-error values returned from `gpiod_to_irq()` can be passed to request_irq() or free_irq(). They will often be stored into IRQ resources for platform devices, by the board-specific initialization code. Note that IRQ trigger options are part of the IRQ interface, e.g. IRQF_TRIGGER_FALLING, as are system wakeup capabilities.

51.4.4 GPIOs and ACPI

On ACPI systems, GPIOs are described by GpioIo()/GpioInt() resources listed by the _CRS configuration objects of devices. Those resources do not provide connection IDs (names) for GPIOs, so it is necessary to use an additional mechanism for this purpose.

Systems compliant with ACPI 5.1 or newer may provide a _DSD configuration object which, among other things, may be used to provide connection IDs for specific GPIOs described by the GpioIo()/GpioInt() resources in _CRS. If that is the case, it will be handled by the GPIO subsystem automatically. However, if the _DSD is not present, the mappings between GpioIo()/GpioInt() resources and GPIO connection IDs need to be provided by device drivers.

For details refer to Documentation/firmware-guide/acpi/gpio-properties.rst

51.4.5 Interacting With the Legacy GPIO Subsystem

Many kernel subsystems still handle GPIOs using the legacy integer-based interface. Although it is strongly encouraged to upgrade them to the safer descriptor-based API, the following two functions allow you to convert a GPIO descriptor into the GPIO integer namespace and vice-versa:

```
int desc_to_gpio(const struct gpio_desc *desc)
struct gpio_desc *gpio_to_desc(unsigned gpio)
```

The GPIO number returned by `desc_to_gpio()` can be safely used as long as the GPIO descriptor has not been freed. All the same, a GPIO number passed to `gpio_to_desc()` must have been properly acquired, and usage of the returned GPIO descriptor is only possible after the GPIO number has been released.

Freeing a GPIO obtained by one API with the other API is forbidden and an unchecked error.

51.5 GPIO Mappings

This document explains how GPIOs can be assigned to given devices and functions.

Note that it only applies to the new descriptor-based interface. For a description of the deprecated integer-based GPIO interface please refer to gpio-legacy.txt (actually, there is no real mapping possible with the old interface; you just fetch an integer from somewhere and request the corresponding GPIO).

All platforms can enable the GPIO library, but if the platform strictly requires GPIO functionality to be present, it needs to select GPIOLIB from its Kconfig. Then, how GPIOs are mapped depends on what the platform uses to describe its hardware layout. Currently, mappings can be defined through device tree, ACPI, and platform data.

51.5.1 Device Tree

GPIOs can easily be mapped to devices and functions in the device tree. The exact way to do it depends on the GPIO controller providing the GPIOs, see the device tree bindings for your controller.

GPIOs mappings are defined in the consumer device's node, in a property named <function>-gpios, where <function> is the function the driver will request through `gpiod_get()`. For example:

```
foo_device {
        compatible = "acme,foo";
        ...
        led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */
                    <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */
                    <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */

        power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;
};
```

Properties named <function>-gpio are also considered valid and old bindings use it but are only supported for compatibility reasons and should not be used for newer bindings since it has been deprecated.

This property will make GPIOs 15, 16 and 17 available to the driver under the "led" function, and GPIO 1 as the "power" GPIO:

```
struct gpio_desc *red, *green, *blue, *power;

red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);

power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```

The led GPIOs will be active high, while the power GPIO will be active low (i.e. gpiod_is_active_low(power) will be true).

The second parameter of the `gpiod_get()` functions, the con_id string, has to be the <function>-prefix of the GPIO suffixes ("gpios" or "gpio", automatically looked up by the gpiod functions internally) used in the device tree. With above "led-gpios" example, use the prefix without the "-" as con_id parameter: "led".

Internally, the GPIO subsystem prefixes the GPIO suffix ("gpios" or "gpio") with the string passed in con_id to get the resulting string (`snprintf(... "%s-%s", con_id, gpio_suffixes[]`).

51.5.2 ACPI

ACPI also supports function names for GPIOs in a similar fashion to DT. The above DT example can be converted to an equivalent ACPI description with the help of _DSD (Device Specific Data), introduced in ACPI 5.1:

```
Device (FOO) {
        Name (_CRS, ResourceTemplate () {
                GpioIo (Exclusive, ..., IoRestrictionOutputOnly,
                        "\\_SB.GPI0") {15} // red
                GpioIo (Exclusive, ..., IoRestrictionOutputOnly,
                        "\\_SB.GPI0") {16} // green
                GpioIo (Exclusive, ..., IoRestrictionOutputOnly,
                        "\\_SB.GPI0") {17} // blue
                GpioIo (Exclusive, ..., IoRestrictionOutputOnly,
                        "\\_SB.GPI0") {1} // power
        })

        Name (_DSD, Package () {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package () {
                        Package () {
                                "led-gpios",
                                Package () {
                                        ^FOO, 0, 0, 1,
                                        ^FOO, 1, 0, 1,
                                        ^FOO, 2, 0, 1,
                                }
                        },
                        Package () {
                                "power-gpios",
                                Package () {^FOO, 3, 0, 0},
                        },
                }
        })
}
```

For more information about the ACPI GPIO bindings see Documentation/firmware-guide/acpi/gpio-properties.rst.

51.5.3 Platform Data

Finally, GPIOs can be bound to devices and functions using platform data. Board files that desire to do so need to include the following header:

```
#include <linux/gpio/machine.h>
```

GPIOs are mapped by the means of tables of lookups, containing instances of the gpiod_lookup structure. Two macros are defined to help declaring such mappings:

```
GPIO_LOOKUP(key, chip_hwnum, con_id, flags)
GPIO_LOOKUP_IDX(key, chip_hwnum, con_id, idx, flags)
```

where

- key is either the label of the gpiod_chip instance providing the GPIO, or the GPIO line name
- chip_hwnum is the hardware number of the GPIO within the chip, or U16_MAX to indicate that key is a GPIO line name
- **con_id is the name of the GPIO function from the device point of view. It** can be NULL, in which case it will match any function.
- idx is the index of the GPIO within the function.
- **flags is defined to specify the following properties:**
  - GPIO_ACTIVE_HIGH - GPIO line is active high
  - GPIO_ACTIVE_LOW - GPIO line is active low
  - GPIO_OPEN_DRAIN - GPIO line is set up as open drain
  - GPIO_OPEN_SOURCE - GPIO line is set up as open source
  - **GPIO_PERSISTENT - GPIO line is persistent during** suspend/resume and maintains its value
  - **GPIO_TRANSITORY - GPIO line is transitory and may loose its** electrical state during suspend/resume

In the future, these flags might be extended to support more properties.

Note that:

1. GPIO line names are not guaranteed to be globally unique, so the first match found will be used.
2. GPIO_LOOKUP() is just a shortcut to GPIO_LOOKUP_IDX() where idx = 0.

A lookup table can then be defined as follows, with an empty entry defining its end. The 'dev_id' field of the table is the identifier of the device that will make use of these GPIOs. It can be NULL, in which case it will be matched for calls to `gpiod_get()` with a NULL device.

```
struct gpiod_lookup_table gpios_table = {
        .dev_id = "foo.0",
        .table = {
                GPIO_LOOKUP_IDX("gpio.0", 15, "led", 0, GPIO_ACTIVE_HIGH),
                GPIO_LOOKUP_IDX("gpio.0", 16, "led", 1, GPIO_ACTIVE_HIGH),
                GPIO_LOOKUP_IDX("gpio.0", 17, "led", 2, GPIO_ACTIVE_HIGH),
                GPIO_LOOKUP("gpio.0", 1, "power", GPIO_ACTIVE_LOW),
                { },
        },
};
```

And the table can be added by the board code as follows:

```
gpiod_add_lookup_table(&gpios_table);
```

The driver controlling "foo.0" will then be able to obtain its GPIOs as follows:

```
struct gpio_desc *red, *green, *blue, *power;

red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);

power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```

Since the "led" GPIOs are mapped as active-high, this example will switch their signals to 1, i.e. enabling the LEDs. And for the "power" GPIO, which is mapped as active-low, its actual signal will be 0 after this code. Contrary to the legacy integer GPIO interface, the active-low property is handled during mapping and is thus transparent to GPIO consumers.

A set of functions such as `gpiod_set_value()` is available to work with the new descriptor-oriented interface.

Boards using platform data can also hog GPIO lines by defining GPIO hog tables.

```
struct gpiod_hog gpio_hog_table[] = {
        GPIO_HOG("gpio.0", 10, "foo", GPIO_ACTIVE_LOW, GPIOD_OUT_HIGH),
        { }
};
```

And the table can be added to the board code as follows:

```
gpiod_add_hogs(gpio_hog_table);
```

The line will be hogged as soon as the gpiochip is created or - in case the chip was created earlier - when the hog table is registered.

51.5.4 Arrays of pins

In addition to requesting pins belonging to a function one by one, a device may also request an array of pins assigned to the function. The way those pins are mapped to the device determines if the array qualifies for fast bitmap processing. If yes, a bitmap is passed over get/set array functions directly between a caller and a respective .get/set_multiple() callback of a GPIO chip.

In order to qualify for fast bitmap processing, the array must meet the following requirements:

- pin hardware number of array member 0 must also be 0,
- pin hardware numbers of consecutive array members which belong to the same chip as member 0 does must also match their array indexes.

Otherwise fast bitmap processing path is not used in order to avoid consecutive pins which belong to the same chip but are not in hardware order being processed separately.

If the array applies for fast bitmap processing path, pins which belong to different chips than member 0 does, as well as those with indexes different from their hardware pin numbers, are excluded from the fast path, both input and output. Moreover, open drain and open source pins are excluded from fast bitmap output processing.

51.6 Subsystem drivers using GPIO

Note that standard kernel drivers exist for common GPIO tasks and will provide the right in-kernel and userspace APIs/ABIs for the job, and that these drivers can quite easily interconnect with other kernel subsystems using hardware descriptions such as device tree or ACPI:

- leds-gpio: drivers/leds/leds-gpio.c will handle LEDs connected to GPIO lines, giving you the LED sysfs interface
- ledtrig-gpio: drivers/leds/trigger/ledtrig-gpio.c will provide a LED trigger, i.e. a LED will turn on/off in response to a GPIO line going high or low (and that LED may in turn use the leds-gpio as per above).
- gpio-keys: drivers/input/keyboard/gpio_keys.c is used when your GPIO line can generate interrupts in response to a key press. Also supports debounce.
- gpio-keys-polled: drivers/input/keyboard/gpio_keys_polled.c is used when your GPIO line cannot generate interrupts, so it needs to be periodically polled by a timer.
- gpio_mouse: drivers/input/mouse/gpio_mouse.c is used to provide a mouse with up to three buttons by simply using GPIOs and no mouse port. You can cut the mouse cable and connect the wires to GPIO lines or solder a mouse connector to the lines for a more permanent solution of this type.
- gpio-beeper: drivers/input/misc/gpio-beeper.c is used to provide a beep from an external speaker connected to a GPIO line.
- extcon-gpio: drivers/extcon/extcon-gpio.c is used when you need to read an external connector status, such as a headset line for an audio driver or an HDMI connector. It will provide a better userspace sysfs interface than GPIO.
- restart-gpio: drivers/power/reset/gpio-restart.c is used to restart/reboot the system by pulling a GPIO line and will register a restart handler so userspace can issue the right system call to restart the system.
- poweroff-gpio: drivers/power/reset/gpio-poweroff.c is used to power the system down by pulling a GPIO line and will register a pm_power_off() callback so that userspace can issue the right system call to power down the system.
- gpio-gate-clock: drivers/clk/clk-gpio.c is used to control a gated clock (off/on) that uses a GPIO, and integrated with the clock subsystem.
- i2c-gpio: drivers/i2c/busses/i2c-gpio.c is used to drive an I2C bus (two wires, SDA and SCL lines) by hammering (bitbang) two GPIO lines. It will appear as any other I2C bus to the system and makes it possible to connect drivers for the I2C devices on the bus like any other I2C bus driver.
- spi_gpio: drivers/spi/spi-gpio.c is used to drive an SPI bus (variable number of wires, at least SCK and optionally MISO, MOSI and chip select lines) using GPIO hammering (bitbang). It will appear as any other SPI bus on the system and makes it possible to connect drivers for SPI devices on the bus like any other SPI bus driver. For example any MMC/SD card can then be connected to this SPI by using the mmc_spi host from the MMC/SD card subsystem.

- w1-gpio: drivers/w1/masters/w1-gpio.c is used to drive a one-wire bus using a GPIO line, integrating with the W1 subsystem and handling devices on the bus like any other W1 device.
- gpio-fan: drivers/hwmon/gpio-fan.c is used to control a fan for cooling the system, connected to a GPIO line (and optionally a GPIO alarm line), presenting all the right in-kernel and sysfs interfaces to make your system not overheat.
- gpio-regulator: drivers/regulator/gpio-regulator.c is used to control a regulator providing a certain voltage by pulling a GPIO line, integrating with the regulator subsystem and giving you all the right interfaces.
- gpio-wdt: drivers/watchdog/gpio_wdt.c is used to provide a watchdog timer that will periodically "ping" a hardware connected to a GPIO line by toggling it from 1-to-0-to-1. If that hardware does not receive its "ping" periodically, it will reset the system.
- gpio-nand: drivers/mtd/nand/raw/gpio.c is used to connect a NAND flash chip to a set of simple GPIO lines: RDY, NCE, ALE, CLE, NWP. It interacts with the NAND flash MTD subsystem and provides chip access and partition parsing like any other NAND driving hardware.
- ps2-gpio: drivers/input/serio/ps2-gpio.c is used to drive a PS/2 (IBM) serio bus, data and clock line, by bit banging two GPIO lines. It will appear as any other serio bus to the system and makes it possible to connect drivers for e.g. keyboards and other PS/2 protocol based devices.
- cec-gpio: drivers/media/platform/cec-gpio/ is used to interact with a CEC Consumer Electronics Control bus using only GPIO. It is used to communicate with devices on the HDMI bus.

Apart from this there are special GPIO drivers in subsystems like MMC/SD to read card detect and write protect GPIO lines, and in the TTY serial subsystem to emulate MCTRL (modem control) signals CTS/RTS by using two GPIO lines. The MTD NOR flash has add-ons for extra GPIO lines too, though the address bus is usually connected directly to the flash.

Use those instead of talking directly to the GPIOs from userspace; they integrate with kernel frameworks better than your userspace code could. Needless to say, just using the appropriate kernel drivers will simplify and speed up your embedded hacking in particular by providing ready-made components.

51.7 Legacy GPIO Interfaces

This provides an overview of GPIO access conventions on Linux.

These calls use the gpio_* naming prefix. No other calls should use that prefix, or the related __gpio_* prefix.

51.7.1 What is a GPIO?

A “General Purpose Input/Output” (GPIO) is a flexible software-controlled digital signal. They are provided from many kinds of chip, and are familiar to Linux developers working with embedded and custom hardware. Each GPIO represents a bit connected to a particular pin, or “ball” on Ball Grid Array (BGA) packages. Board schematics show which external hardware connects to which GPIOs. Drivers can be written generically, so that board setup code passes such pin configuration data to drivers.

System-on-Chip (SOC) processors heavily rely on GPIOs. In some cases, every non-dedicated pin can be configured as a GPIO; and most chips have at least several dozen of them. Programmable logic devices (like FPGAs) can easily provide GPIOs; multifunction chips like power managers, and audio codecs often have a few such pins to help with pin scarcity on SOCs; and there are also “GPIO Expander” chips that connect using the I2C or SPI serial busses. Most PC southbridges have a few dozen GPIO-capable pins (with only the BIOS firmware knowing how they’re used).

The exact capabilities of GPIOs vary between systems. Common options:

- Output values are writable (high=1, low=0). Some chips also have options about how that value is driven, so that for example only one value might be driven ⋯supporting “wire-OR” and similar schemes for the other value (notably, “open drain” signaling).
- Input values are likewise readable (1, 0). Some chips support readback of pins configured as “output”, which is very useful in such “wire-OR” cases (to support bidirectional signaling). GPIO controllers may have input deglitch/debounce logic, sometimes with software controls.
- Inputs can often be used as IRQ signals, often edge triggered but sometimes level triggered. Such IRQs may be configurable as system wakeup events, to wake the system from a low power state.
- Usually a GPIO will be configurable as either input or output, as needed by different product boards; single direction ones exist too.
- Most GPIOs can be accessed while holding spinlocks, but those accessed through a serial bus normally can’t. Some systems support both types.

On a given board each GPIO is used for one specific purpose like monitoring MMC/SD card insertion/removal, detecting card writeprotect status, driving a LED, configuring a transceiver, bitbanging a serial bus, poking a hardware watchdog, sensing a switch, and so on.

51.7.2 GPIO conventions

Note that this is called a "convention" because you don't need to do it this way, and it's no crime if you don't. There **are** cases where portability is not the main issue; GPIOs are often used for the kind of board-specific glue logic that may even change between board revisions, and can't ever be used on a board that's wired differently. Only least-common-denominator functionality can be very portable. Other features are platform-specific, and that can be critical for glue logic.

Plus, this doesn't require any implementation framework, just an interface. One platform might implement it as simple inline functions accessing chip registers; another might implement it by delegating through abstractions used for several very different kinds of GPIO controller. (There is some optional code supporting such an implementation strategy, described later in this document, but drivers acting as clients to the GPIO interface must not care how it's implemented.)

That said, if the convention is supported on their platform, drivers should use it when possible. Platforms must select GPIOLIB if GPIO functionality is strictly required. Drivers that can't work without standard GPIO calls should have Kconfig entries which depend on GPIOLIB. The GPIO calls are available, either as "real code" or as optimized-away stubs, when drivers use the include file:

```
#include <linux/gpio.h>
```

If you stick to this convention then it'll be easier for other developers to see what your code is doing, and help maintain it.

Note that these operations include I/O barriers on platforms which need to use them; drivers don't need to add them explicitly.

Identifying GPIOs

GPIOs are identified by unsigned integers in the range 0..MAX_INT. That reserves "negative" numbers for other purposes like marking signals as "not available on this board", or indicating faults. Code that doesn't touch the underlying hardware treats these integers as opaque cookies.

Platforms define how they use those integers, and usually #define symbols for the GPIO lines so that board-specific setup code directly corresponds to the relevant schematics. In contrast, drivers should only use GPIO numbers passed to them from that setup code, using platform_data to hold board-specific pin configuration data (along with other board specific data they need). That avoids portability problems.

So for example one platform uses numbers 32-159 for GPIOs; while another uses numbers 0..63 with one set of GPIO controllers, 64-79 with another type of GPIO controller, and on one particular board 80-95 with an FPGA. The numbers need not be contiguous; either of those platforms could also use numbers 2000-2063 to identify GPIOs in a bank of I2C GPIO expanders.

If you want to initialize a structure with an invalid GPIO number, use some negative number (perhaps "-EINVAL"); that will never be valid. To test if such number from such a structure could reference a GPIO, you may use this predicate:

```
int gpio_is_valid(int number);
```

A number that's not valid will be rejected by calls which may request or free GPIOs (see below). Other numbers may also be rejected; for example, a number might be valid but temporarily unused on a given board.

Whether a platform supports multiple GPIO controllers is a platform-specific implementation issue, as are whether that support can leave "holes" in the space of GPIO numbers, and whether new controllers can be added at runtime. Such issues can affect things including whether adjacent GPIO numbers are both valid.

Using GPIOs

The first thing a system should do with a GPIO is allocate it, using the gpio_request() call; see later.

One of the next things to do with a GPIO, often in board setup code when setting up a platform_device using the GPIO, is mark its direction:

```
/* set as input or output, returning 0 or negative errno */
int gpio_direction_input(unsigned gpio);
int gpio_direction_output(unsigned gpio, int value);
```

The return value is zero for success, else a negative errno. It should be checked, since the get/set calls don't have error returns and since misconfiguration is possible. You should normally issue these calls from a task context. However, for spinlock-safe GPIOs it's OK to use them before tasking is enabled, as part of early board setup.

For output GPIOs, the value provided becomes the initial output value. This helps avoid signal glitching during system startup.

For compatibility with legacy interfaces to GPIOs, setting the direction of a GPIO implicitly requests that GPIO (see below) if it has not been requested already. That compatibility is being removed from the optional gpiolib framework.

Setting the direction can fail if the GPIO number is invalid, or when that particular GPIO can't be used in that mode. It's generally a bad idea to rely on boot firmware to have set the direction correctly, since it probably wasn't validated to do more than boot Linux. (Similarly, that board setup code probably needs to multiplex that pin as a GPIO, and configure pullups/pulldowns appropriately.)

Spinlock-Safe GPIO access

Most GPIO controllers can be accessed with memory read/write instructions. Those don't need to sleep, and can safely be done from inside hard (nonthreaded) IRQ handlers and similar contexts.

Use the following calls to access such GPIOs, for which gpio_cansleep() will always return false (see below):

```
/* GPIO INPUT:  return zero or nonzero */
int gpio_get_value(unsigned gpio);

/* GPIO OUTPUT */
void gpio_set_value(unsigned gpio, int value);
```

The values are boolean, zero for low, nonzero for high. When reading the value of an output pin, the value returned should be what's seen on the pin ···that won't always match the specified output value, because of issues including open-drain signaling and output latencies.

The get/set calls have no error returns because "invalid GPIO" should have been reported earlier from gpio_direction_*(). However, note that not all platforms can read the value of output pins; those that can't should always return zero. Also, using these calls for GPIOs that can't safely be accessed without sleeping (see below) is an error.

Platform-specific implementations are encouraged to optimize the two calls to access the GPIO value in cases where the GPIO number (and for output, value) are constant. It's normal for them to need only a couple of instructions in such cases (reading or writing a hardware register), and not to need spinlocks. Such optimized calls can make bitbanging applications a lot more efficient (in both space and time) than spending dozens of instructions on subroutine calls.

GPIO access that may sleep

Some GPIO controllers must be accessed using message based busses like I2C or SPI. Commands to read or write those GPIO values require waiting to get to the head of a queue to transmit a command and get its response. This requires sleeping, which can't be done from inside IRQ handlers.

Platforms that support this type of GPIO distinguish them from other GPIOs by returning nonzero from this call (which requires a valid GPIO number, which should have been previously allocated with gpio_request):

```
int gpio_cansleep(unsigned gpio);
```

To access such GPIOs, a different set of accessors is defined:

```
/* GPIO INPUT:  return zero or nonzero, might sleep */
int gpio_get_value_cansleep(unsigned gpio);

/* GPIO OUTPUT, might sleep */
void gpio_set_value_cansleep(unsigned gpio, int value);
```

Accessing such GPIOs requires a context which may sleep, for example a threaded IRQ handler, and those accessors must be used instead of spinlock-safe accessors without the cansleep() name suffix.

Other than the fact that these accessors might sleep, and will work on GPIOs that can't be accessed from hardIRQ handlers, these calls act the same as the spinlock-safe calls.

IN ADDITION calls to setup and configure such GPIOs must be made from contexts which may sleep, since they may need to access the GPIO controller chip too (These setup calls are usually made from board setup or driver probe/teardown code, so this is an easy constraint.):

```
        gpio_direction_input()
        gpio_direction_output()
```

(continues on next page)

(continued from previous page)

```
        gpio_request()

##      gpio_request_one()
##      gpio_request_array()
##      gpio_free_array()

        gpio_free()
        gpio_set_debounce()
```

Claiming and Releasing GPIOs

To help catch system configuration errors, two calls are defined:

```
/* request GPIO, returning 0 or negative errno.
 * non-null labels may be useful for diagnostics.
 */
int gpio_request(unsigned gpio, const char *label);

/* release previously-claimed GPIO */
void gpio_free(unsigned gpio);
```

Passing invalid GPIO numbers to gpio_request() will fail, as will requesting GPIOs that have already been claimed with that call. The return value of gpio_request() must be checked. You should normally issue these calls from a task context. However, for spinlock-safe GPIOs it' s OK to request GPIOs before tasking is enabled, as part of early board setup.

These calls serve two basic purposes. One is marking the signals which are actually in use as GPIOs, for better diagnostics; systems may have several hundred potential GPIOs, but often only a dozen are used on any given board. Another is to catch conflicts, identifying errors when (a) two or more drivers wrongly think they have exclusive use of that signal, or (b) something wrongly believes it' s safe to remove drivers needed to manage a signal that' s in active use. That is, requesting a GPIO can serve as a kind of lock.

Some platforms may also use knowledge about what GPIOs are active for power management, such as by powering down unused chip sectors and, more easily, gating off unused clocks.

For GPIOs that use pins known to the pinctrl subsystem, that subsystem should be informed of their use; a gpiolib driver' s .request() operation may call pinctrl_gpio_request(), and a gpiolib driver' s .free() operation may call pinctrl_gpio_free(). The pinctrl subsystem allows a pinctrl_gpio_request() to succeed concurrently with a pin or pingroup being "owned" by a device for pin multiplexing.

Any programming of pin multiplexing hardware that is needed to route the GPIO signal to the appropriate pin should occur within a GPIO driver' s .direction_input() or .direction_output() operations, and occur after any setup of an output GPIO' s value. This allows a glitch-free migration from a pin' s special function to GPIO. This is sometimes required when using a GPIO to implement a workaround on signals typically driven by a non-GPIO HW block.

Some platforms allow some or all GPIO signals to be routed to different pins.

Similarly, other aspects of the GPIO or pin may need to be configured, such as pullup/pulldown. Platform software should arrange that any such details are configured prior to gpio_request() being called for those GPIOs, e.g. using the pinctrl subsystem's mapping table, so that GPIO users need not be aware of these details.

Also note that it's your responsibility to have stopped using a GPIO before you free it.

Considering in most cases GPIOs are actually configured right after they are claimed, three additional calls are defined:

```
/* request a single GPIO, with initial configuration specified by
 * 'flags', identical to gpio_request() wrt other arguments and
 * return value
 */
int gpio_request_one(unsigned gpio, unsigned long flags, const char␣
↪*label);

/* request multiple GPIOs in a single call
 */
int gpio_request_array(struct gpio *array, size_t num);

/* release multiple GPIOs in a single call
 */
void gpio_free_array(struct gpio *array, size_t num);
```

where 'flags' is currently defined to specify the following properties:

- GPIOF_DIR_IN - to configure direction as input
- GPIOF_DIR_OUT - to configure direction as output
- GPIOF_INIT_LOW - as output, set initial level to LOW
- GPIOF_INIT_HIGH - as output, set initial level to HIGH
- GPIOF_OPEN_DRAIN - gpio pin is open drain type.
- GPIOF_OPEN_SOURCE - gpio pin is open source type.
- GPIOF_EXPORT_DIR_FIXED - export gpio to sysfs, keep direction
- GPIOF_EXPORT_DIR_CHANGEABLE - also export, allow changing direction

since GPIOF_INIT_* are only valid when configured as output, so group valid combinations as:

- GPIOF_IN - configure as input
- GPIOF_OUT_INIT_LOW - configured as output, initial level LOW
- GPIOF_OUT_INIT_HIGH - configured as output, initial level HIGH

When setting the flag as GPIOF_OPEN_DRAIN then it will assume that pins is open drain type. Such pins will not be driven to 1 in output mode. It is require to connect pull-up on such pins. By enabling this flag, gpio lib will make the direction to input when it is asked to set value of 1 in output mode to make the pin HIGH. The pin is make to LOW by driving value 0 in output mode.

When setting the flag as GPIOF_OPEN_SOURCE then it will assume that pins is open source type. Such pins will not be driven to 0 in output mode. It is require

to connect pull-down on such pin. By enabling this flag, gpio lib will make the direction to input when it is asked to set value of 0 in output mode to make the pin LOW. The pin is make to HIGH by driving value 1 in output mode.

In the future, these flags can be extended to support more properties.

Further more, to ease the claim/release of multiple GPIOs, 'struct gpio' is introduced to encapsulate all three fields as:

```
struct gpio {
        unsigned        gpio;
        unsigned long   flags;
        const char      *label;
};
```

A typical example of usage:

```
static struct gpio leds_gpios[] = {
        { 32, GPIOF_OUT_INIT_HIGH, "Power LED" }, /* default to ON */
        { 33, GPIOF_OUT_INIT_LOW,  "Green LED" }, /* default to OFF */
        { 34, GPIOF_OUT_INIT_LOW,  "Red LED"   }, /* default to OFF */
        { 35, GPIOF_OUT_INIT_LOW,  "Blue LED"  }, /* default to OFF */
        { ... },
};

err = gpio_request_one(31, GPIOF_IN, "Reset Button");
if (err)
        ...

err = gpio_request_array(leds_gpios, ARRAY_SIZE(leds_gpios));
if (err)
        ...

gpio_free_array(leds_gpios, ARRAY_SIZE(leds_gpios));
```

GPIOs mapped to IRQs

GPIO numbers are unsigned integers; so are IRQ numbers. These make up two logically distinct namespaces (GPIO 0 need not use IRQ 0). You can map between them using calls like:

```
/* map GPIO numbers to IRQ numbers */
int gpio_to_irq(unsigned gpio);

/* map IRQ numbers to GPIO numbers (avoid using this) */
int irq_to_gpio(unsigned irq);
```

Those return either the corresponding number in the other namespace, or else a negative errno code if the mapping can't be done. (For example, some GPIOs can't be used as IRQs.) It is an unchecked error to use a GPIO number that wasn't set up as an input using gpio_direction_input(), or to use an IRQ number that didn't originally come from gpio_to_irq().

These two mapping calls are expected to cost on the order of a single addition or subtraction. They're not allowed to sleep.

Non-error values returned from gpio_to_irq() can be passed to request_irq() or free_irq(). They will often be stored into IRQ resources for platform devices, by the board-specific initialization code. Note that IRQ trigger options are part of the IRQ interface, e.g. IRQF_TRIGGER_FALLING, as are system wakeup capabilities.

Non-error values returned from irq_to_gpio() would most commonly be used with gpio_get_value(), for example to initialize or update driver state when the IRQ is edge-triggered. Note that some platforms don't support this reverse mapping, so you should avoid using it.

Emulating Open Drain Signals

Sometimes shared signals need to use "open drain" signaling, where only the low signal level is actually driven. (That term applies to CMOS transistors; "open collector" is used for TTL.) A pullup resistor causes the high signal level. This is sometimes called a "wire-AND"; or more practically, from the negative logic (low=true) perspective this is a "wire-OR".

One common example of an open drain signal is a shared active-low IRQ line. Also, bidirectional data bus signals sometimes use open drain signals.

Some GPIO controllers directly support open drain outputs; many don't. When you need open drain signaling but your hardware doesn't directly support it, there's a common idiom you can use to emulate it with any GPIO pin that can be used as either an input or an output:

LOW: gpio_direction_output(gpio, 0) ···this drives the signal and overrides the pullup.

HIGH: gpio_direction_input(gpio) ···this turns off the output, so the pullup (or some other device) controls the signal.

If you are "driving" the signal high but gpio_get_value(gpio) reports a low value (after the appropriate rise time passes), you know some other component is driving the shared signal low. That's not necessarily an error. As one common example, that's how I2C clocks are stretched: a slave that needs a slower clock delays the rising edge of SCK, and the I2C master adjusts its signaling rate accordingly.

GPIO controllers and the pinctrl subsystem

A GPIO controller on a SOC might be tightly coupled with the pinctrl subsystem, in the sense that the pins can be used by other functions together with an optional gpio feature. We have already covered the case where e.g. a GPIO controller need to reserve a pin or set the direction of a pin by calling any of:

```
pinctrl_gpio_request()
pinctrl_gpio_free()
pinctrl_gpio_direction_input()
pinctrl_gpio_direction_output()
```

But how does the pin control subsystem cross-correlate the GPIO numbers (which are a global business) to a certain pin on a certain pin controller?

This is done by registering "ranges" of pins, which are essentially cross-reference tables. These are described in Documentation/driver-api/pinctl.rst

While the pin allocation is totally managed by the pinctrl subsystem, gpio (under gpiolib) is still maintained by gpio drivers. It may happen that different pin ranges in a SoC is managed by different gpio drivers.

This makes it logical to let gpio drivers announce their pin ranges to the pin ctrl subsystem before it will call 'pinctrl_gpio_request' in order to request the corresponding pin to be prepared by the pinctrl subsystem before any gpio usage.

For this, the gpio controller can register its pin range with pinctrl subsystem. There are two ways of doing it currently: with or without DT.

For with DT support refer to Documentation/devicetree/bindings/gpio/gpio.txt.

For non-DT support, user can call `gpiochip_add_pin_range()` with appropriate parameters to register a range of gpio pins with a pinctrl driver. For this exact name string of pinctrl device has to be passed as one of the argument to this routine.

51.7.3 What do these conventions omit?

One of the biggest things these conventions omit is pin multiplexing, since this is highly chip-specific and nonportable. One platform might not need explicit multiplexing; another might have just two options for use of any given pin; another might have eight options per pin; another might be able to route a given GPIO to any one of several pins. (Yes, those examples all come from systems that run Linux today.)

Related to multiplexing is configuration and enabling of the pullups or pulldowns integrated on some platforms. Not all platforms support them, or support them in the same way; and any given board might use external pullups (or pulldowns) so that the on-chip ones should not be used. (When a circuit needs 5 kOhm, on-chip 100 kOhm resistors won't do.) Likewise drive strength (2 mA vs 20 mA) and voltage (1.8V vs 3.3V) is a platform-specific issue, as are models like (not) having a one-to-one correspondence between configurable pins and GPIOs.

There are other system-specific mechanisms that are not specified here, like the aforementioned options for input de-glitching and wire-OR output. Hardware may support reading or writing GPIOs in gangs, but that's usually configuration dependent: for GPIOs sharing the same bank. (GPIOs are commonly grouped in banks of 16 or 32, with a given SOC having several such banks.) Some systems can trigger IRQs from output GPIOs, or read values from pins not managed as GPIOs. Code relying on such mechanisms will necessarily be nonportable.

Dynamic definition of GPIOs is not currently standard; for example, as a side effect of configuring an add-on board with some GPIO expanders.

51.7.4 GPIO implementor's framework (OPTIONAL)

As noted earlier, there is an optional implementation framework making it easier for platforms to support different kinds of GPIO controller using the same programming interface. This framework is called "gpiolib".

As a debugging aid, if debugfs is available a /sys/kernel/debug/gpio file will be found there. That will list all the controllers registered through this framework, and the state of the GPIOs currently in use.

Controller Drivers: gpio_chip

In this framework each GPIO controller is packaged as a "struct gpio_chip" with information common to each controller of that type:

- methods to establish GPIO direction
- methods used to access GPIO values
- flag saying whether calls to its methods may sleep
- optional debugfs dump method (showing extra state like pullup config)
- label for diagnostics

There is also per-instance data, which may come from device.platform_data: the number of its first GPIO, and how many GPIOs it exposes.

The code implementing a gpio_chip should support multiple instances of the controller, possibly using the driver model. That code will configure each gpio_chip and issue gpiochip_add(). Removing a GPIO controller should be rare; use `gpiochip_remove()` when it is unavoidable.

Most often a gpio_chip is part of an instance-specific structure with state not exposed by the GPIO interfaces, such as addressing, power management, and more. Chips such as codecs will have complex non-GPIO state.

Any debugfs dump method should normally ignore signals which haven't been requested as GPIOs. They can use `gpiochip_is_requested()`, which returns either NULL or the label associated with that GPIO when it was requested.

Platform Support

To force-enable this framework, a platform's Kconfig will "select" GPIOLIB, else it is up to the user to configure support for GPIO.

It may also provide a custom value for ARCH_NR_GPIOS, so that it better reflects the number of GPIOs in actual use on that platform, without wasting static table space. (It should count both built-in/SoC GPIOs and also ones on GPIO expanders.

If neither of these options are selected, the platform does not support GPIOs through GPIO-lib and the code cannot be enabled by the user.

Trivial implementations of those functions can directly use framework code, which always dispatches through the gpio_chip:

```
#define gpio_get_value        __gpio_get_value
#define gpio_set_value        __gpio_set_value
#define gpio_cansleep         __gpio_cansleep
```

Fancier implementations could instead define those as inline functions with logic optimizing access to specific SOC-based GPIOs. For example, if the referenced GPIO is the constant “12”, getting or setting its value could cost as little as two or three instructions, never sleeping. When such an optimization is not possible those calls must delegate to the framework code, costing at least a few dozen instructions. For bitbanged I/O, such instruction savings can be significant.

For SOCs, platform-specific code defines and registers gpio_chip instances for each bank of on-chip GPIOs. Those GPIOs should be numbered/labeled to match chip vendor documentation, and directly match board schematics. They may well start at zero and go up to a platform-specific limit. Such GPIOs are normally integrated into platform initialization to make them always be available, from arch_initcall() or earlier; they can often serve as IRQs.

Board Support

For external GPIO controllers – such as I2C or SPI expanders, ASICs, multi function devices, FPGAs or CPLDs – most often board-specific code handles registering controller devices and ensures that their drivers know what GPIO numbers to use with gpiochip_add(). Their numbers often start right after platform-specific GPIOs.

For example, board setup code could create structures identifying the range of GPIOs that chip will expose, and passes them to each GPIO expander chip using platform_data. Then the chip driver’s probe() routine could pass that data to gpiochip_add().

Initialization order can be important. For example, when a device relies on an I2C-based GPIO, its probe() routine should only be called after that GPIO becomes available. That may mean the device should not be registered until calls for that GPIO can work. One way to address such dependencies is for such gpio_chip controllers to provide setup() and teardown() callbacks to board specific code; those board specific callbacks would register devices once all the necessary resources are available, and remove them later when the GPIO controller device becomes unavailable.

51.7.5 Sysfs Interface for Userspace (OPTIONAL)

Platforms which use the “gpiolib” implementors framework may choose to configure a sysfs user interface to GPIOs. This is different from the debugfs interface, since it provides control over GPIO direction and value instead of just showing a gpio state summary. Plus, it could be present on production systems without debugging support.

Given appropriate hardware documentation for the system, userspace could know for example that GPIO #23 controls the write protect line used to protect boot loader segments in flash memory. System upgrade procedures may need to temporarily remove that protection, first importing a GPIO, then changing its output state, then updating the code before re-enabling the write protection. In normal

use, GPIO #23 would never be touched, and the kernel would have no need to know about it.

Again depending on appropriate hardware documentation, on some systems userspace GPIO can be used to determine system configuration data that standard kernels won't know about. And for some tasks, simple userspace GPIO drivers could be all that the system really needs.

Note that standard kernel drivers exist for common "LEDs and Buttons"GPIO tasks: "leds-gpio" and "gpio_keys", respectively. Use those instead of talking directly to the GPIOs; they integrate with kernel frameworks better than your userspace code could.

Paths in Sysfs

There are three kinds of entry in /sys/class/gpio:

- Control interfaces used to get userspace control over GPIOs;
- GPIOs themselves; and
- GPIO controllers ("gpio_chip" instances).

That's in addition to standard files including the "device" symlink.

The control interfaces are write-only:

/sys/class/gpio/

"export" ···Userspace may ask the kernel to export control of a GPIO to userspace by writing its number to this file.

Example: "echo 19 > export" will create a "gpio19" node for GPIO #19, if that's not requested by kernel code.

"unexport" ···Reverses the effect of exporting to userspace.

Example: "echo 19 > unexport" will remove a "gpio19" node exported using the "export" file.

GPIO signals have paths like /sys/class/gpio/gpio42/ (for GPIO #42) and have the following read/write attributes:

/sys/class/gpio/gpioN/

"direction" ···reads as either "in" or "out". This value may normally be written. Writing as "out" defaults to initializing the value as low. To ensure glitch free operation, values "low" and "high" may be written to configure the GPIO as an output with that initial value.

Note that this attribute will not exist if the kernel doesn't support changing the direction of a GPIO, or it was exported by kernel code that didn't explicitly allow userspace to re-configure this GPIO's direction.

"value" ···reads as either 0 (low) or 1 (high). If the GPIO is configured as an output, this value may be written; any nonzero value is treated as high.

If the pin can be configured as interrupt-generating interrupt and if it has been configured to generate interrupts (see the description of "edge"), you can poll(2) on that file and poll(2) will return whenever the interrupt was triggered. If you use poll(2), set the events POLLPRI. If you use select(2), set the file descriptor in exceptfds. After poll(2) returns, either lseek(2) to the beginning of the sysfs file and read the new value or close the file and re-open it to read the value.

"edge" ···reads as either "none", "rising", "falling", or "both". Write these strings to select the signal edge(s) that will make poll(2) on the "value" file return.

This file exists only if the pin can be configured as an interrupt generating input pin.

"active_low" ···reads as either 0 (false) or 1 (true). Write any nonzero value to invert the value attribute both for reading and writing. Existing and subsequent poll(2) support configuration via the edge attribute for "rising" and "falling" edges will follow this setting.

GPIO controllers have paths like /sys/class/gpio/gpiochip42/ (for the controller implementing GPIOs starting at #42) and have the following read-only attributes:

/sys/class/gpio/gpiochipN/

"base" ···same as N, the first GPIO managed by this chip

"label" ···provided for diagnostics (not always unique)

"ngpio" ···how many GPIOs this manges (N to N + ngpio - 1)

Board documentation should in most cases cover what GPIOs are used for what purposes. However, those numbers are not always stable; GPIOs on a daughtercard might be different depending on the base board being used, or other cards in the stack. In such cases, you may need to use the gpiochip nodes (possibly in conjunction with schematics) to determine the correct GPIO number to use for a given signal.

Exporting from Kernel code

Kernel code can explicitly manage exports of GPIOs which have already been requested using gpio_request():

```
/* export the GPIO to userspace */
int gpio_export(unsigned gpio, bool direction_may_change);

/* reverse gpio_export() */
void gpio_unexport();

/* create a sysfs link to an exported GPIO node */
int gpio_export_link(struct device *dev, const char *name,
        unsigned gpio)
```

After a kernel driver requests a GPIO, it may only be made available in the sysfs interface by gpio_export(). The driver can control whether the signal direction may change. This helps drivers prevent userspace code from accidentally clobbering important system state.

This explicit exporting can help with debugging (by making some kinds of experiments easier), or can provide an always-there interface that's suitable for documenting as part of a board support package.

After the GPIO has been exported, gpio_export_link() allows creating symlinks from elsewhere in sysfs to the GPIO sysfs node. Drivers can use this to provide the interface under their own device in sysfs with a descriptive name.

51.7.6 API Reference

The functions listed in this section are deprecated. The GPIO descriptor based API should be used in new code.

int **gpio_request_one**(unsigned gpio, unsigned long flags, const char * label)
request a single GPIO with initial configuration

Parameters

unsigned gpio the GPIO number

unsigned long flags GPIO configuration as specified by GPIOF_*

const char * label a literal description string of this GPIO

int **gpio_request_array**(const struct gpio * array, size_t num)
request multiple GPIOs in a single call

Parameters

const struct gpio * array array of the 'struct gpio'

size_t num how many GPIOs in the array

void **gpio_free_array**(const struct gpio * array, size_t num)
release multiple GPIOs in a single call

Parameters

const struct gpio * array array of the 'struct gpio'

size_t num how many GPIOs in the array

51.8 A driver for a selfmade cheap BT8xx based PCI GPIO-card (bt8xxgpio)

For advanced documentation, see https://bues.ch/cms/unmaintained/btgpio.html

A generic digital 24-port PCI GPIO card can be built out of an ordinary Brooktree bt848, bt849, bt878 or bt879 based analog TV tuner card. The Brooktree chip is used in old analog Hauppauge WinTV PCI cards. You can easily find them used for low prices on the net.

The bt8xx chip does have 24 digital GPIO ports. These ports are accessible via 24 pins on the SMD chip package.

51.8.1 How to physically access the GPIO pins

The are several ways to access these pins. One might unsolder the whole chip and put it on a custom PCI board, or one might only unsolder each individual GPIO pin and solder that to some tiny wire. As the chip package really is tiny there are some advanced soldering skills needed in any case.

The physical pinouts are drawn in the following ASCII art. The GPIO pins are marked with G00-G23:

```
                                  G G G G G G G G G G G G     G G G G G G
                                  0 0 0 0 0 0 0 0 0 0 1 1     1 1 1 1 1 1
                                  0 1 2 3 4 5 6 7 8 9 0 1     2 3 4 5 6 7
  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
 | |
  ------------------------------------------------------------------------
--
--|                                  ^                                   ^
  |--
--|                                  pin 86                               pin 67
  |--
--|                                                                       
  |--
--|                                                                     pin 61 >
  |-- G18
--|                                                                       
  |-- G19
--|                                                                       
  |-- G20
--|                                                                       
  |-- G21
--|                                                                       
  |-- G22
--|                                                                     pin 56 >
  |-- G23
--|                                                                       
  |--
--|                               Brooktree 878/879                       
  |--
--|                                                                       
  |--
--|                                                                       
  |--
--|                                                                       
  |--
--|                                                                       
  |--
--|                                                                       
  |--
--|                                                                       
  |--
--|                                                                       
  |--
```

(continues on next page)

(continued from previous page)

```
--|                                                                            ␣
↪ |--
--|                                                                            ␣
↪ |--
--|                                                                            ␣
↪ |--
--|                                                                            ␣
↪ |--
--|                                                                            ␣
↪ |--
--|                                                                            ␣
↪ |--
--|   0                                                                        ␣
↪ |--
--|                                                                            ␣
↪ |--
  ------------------------------------------------------------------------------
↪--
  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |␣
↪| |
  ^
  This is pin 1
```

51.9 Core

struct **gpio_irq_chip**
GPIO interrupt controller

Definition

```
struct gpio_irq_chip {
  struct irq_chip *chip;
  struct irq_domain *domain;
  const struct irq_domain_ops *domain_ops;
#ifdef CONFIG_IRQ_DOMAIN_HIERARCHY;
  struct fwnode_handle *fwnode;
  struct irq_domain *parent_domain;
  int (*child_to_parent_hwirq)(struct gpio_chip *gc,unsigned int child_
↪hwirq,unsigned int child_type,unsigned int *parent_hwirq, unsigned int␣
↪*parent_type);
  void *(*populate_parent_alloc_arg)(struct gpio_chip *gc,unsigned int␣
↪parent_hwirq, unsigned int parent_type);
  unsigned int (*child_offset_to_irq)(struct gpio_chip *gc, unsigned int␣
↪pin);
  struct irq_domain_ops child_irq_domain_ops;
#endif;
  irq_flow_handler_t handler;
  unsigned int default_type;
  struct lock_class_key *lock_key;
  struct lock_class_key *request_key;
  irq_flow_handler_t parent_handler;
  void *parent_handler_data;
  unsigned int num_parents;
```

(continues on next page)

(continued from previous page)

```
  unsigned int *parents;
  unsigned int *map;
  bool threaded;
  int (*init_hw)(struct gpio_chip *gc);
  void (*init_valid_mask)(struct gpio_chip *gc,unsigned long *valid_mask,␣
↪unsigned int ngpios);
  unsigned long *valid_mask;
  unsigned int first;
  void (*irq_enable)(struct irq_data *data);
  void (*irq_disable)(struct irq_data *data);
  void (*irq_unmask)(struct irq_data *data);
  void (*irq_mask)(struct irq_data *data);
};
```

Members

chip GPIO IRQ chip implementation, provided by GPIO driver.

domain Interrupt translation domain; responsible for mapping between GPIO hwirq number and Linux IRQ number.

domain_ops Table of interrupt domain operations for this IRQ chip.

fwnode Firmware node corresponding to this gpiochip/irqchip, necessary for hierarchical irqdomain support.

parent_domain If non-NULL, will be set as the parent of this GPIO interrupt controller's IRQ domain to establish a hierarchical interrupt domain. The presence of this will activate the hierarchical interrupt support.

child_to_parent_hwirq This callback translates a child hardware IRQ offset to a parent hardware IRQ offset on a hierarchical interrupt chip. The child hardware IRQs correspond to the GPIO index 0..ngpio-1 (see the ngpio field of struct gpio_chip) and the corresponding parent hardware IRQ and type (such as IRQ_TYPE_*) shall be returned by the driver. The driver can calculate this from an offset or using a lookup table or whatever method is best for this chip. Return 0 on successful translation in the driver.

If some ranges of hardware IRQs do not have a corresponding parent HWIRQ, return -EINVAL, but also make sure to fill in **valid_mask** and **need_valid_mask** to make these GPIO lines unavailable for translation.

populate_parent_alloc_arg This optional callback allocates and populates the specific struct for the parent's IRQ domain. If this is not specified, then `gpiochip_populate_parent_fwspec_twocell` will be used. A four-cell variant named `gpiochip_populate_parent_fwspec_fourcell` is also available.

child_offset_to_irq This optional callback is used to translate the child's GPIO line offset on the GPIO chip to an IRQ number for the GPIO to_irq() callback. If this is not specified, then a default callback will be provided that returns the line offset.

child_irq_domain_ops The IRQ domain operations that will be used for this GPIO IRQ chip. If no operations are provided, then default callbacks will be populated to setup the IRQ hierarchy. Some drivers need to supply their own translate function.

handler The IRQ handler to use (often a predefined IRQ core function) for GPIO IRQs, provided by GPIO driver.

default_type Default IRQ triggering type applied during GPIO driver initialization, provided by GPIO driver.

lock_key Per GPIO IRQ chip lockdep class for IRQ lock.

request_key Per GPIO IRQ chip lockdep class for IRQ request.

parent_handler The interrupt handler for the GPIO chip's parent interrupts, may be NULL if the parent interrupts are nested rather than cascaded.

parent_handler_data Data associated, and passed to, the handler for the parent interrupt.

num_parents The number of interrupt parents of a GPIO chip.

parents A list of interrupt parents of a GPIO chip. This is owned by the driver, so the core will only reference this list, not modify it.

map A list of interrupt parents for each line of a GPIO chip.

threaded True if set the interrupt handling uses nested threads.

init_hw optional routine to initialize hardware before an IRQ chip will be added. This is quite useful when a particular driver wants to clear IRQ related registers in order to avoid undesired events.

init_valid_mask optional routine to initialize **valid_mask**, to be used if not all GPIO lines are valid interrupts. Sometimes some lines just cannot fire interrupts, and this routine, when defined, is passed a bitmap in "valid_mask" and it will have ngpios bits from 0..(ngpios-1) set to "1" as in valid. The callback can then directly set some bits to "0" if they cannot be used for interrupts.

valid_mask If not `NULL` holds bitmask of GPIOs which are valid to be included in IRQ domain of the chip.

first Required for static IRQ allocation. If set, irq_domain_add_simple() will allocate and map all IRQs during initialization.

irq_enable Store old irq_chip irq_enable callback

irq_disable Store old irq_chip irq_disable callback

irq_unmask Store old irq_chip irq_unmask callback

irq_mask Store old irq_chip irq_mask callback

struct **gpio_chip**
abstract a GPIO controller

Definition

```
struct gpio_chip {
  const char              *label;
  struct gpio_device      *gpiodev;
  struct device           *parent;
  struct module           *owner;
  int (*request)(struct gpio_chip *gc, unsigned int offset);
  void (*free)(struct gpio_chip *gc, unsigned int offset);
```

(continues on next page)

(continued from previous page)

```
  int (*get_direction)(struct gpio_chip *gc, unsigned int offset);
  int (*direction_input)(struct gpio_chip *gc, unsigned int offset);
  int (*direction_output)(struct gpio_chip *gc, unsigned int offset, int␣
↪value);
  int (*get)(struct gpio_chip *gc, unsigned int offset);
  int (*get_multiple)(struct gpio_chip *gc,unsigned long *mask, unsigned␣
↪long *bits);
  void (*set)(struct gpio_chip *gc, unsigned int offset, int value);
  void (*set_multiple)(struct gpio_chip *gc,unsigned long *mask, unsigned␣
↪long *bits);
  int (*set_config)(struct gpio_chip *gc,unsigned int offset, unsigned␣
↪long config);
  int (*to_irq)(struct gpio_chip *gc, unsigned int offset);
  void (*dbg_show)(struct seq_file *s, struct gpio_chip *gc);
  int (*init_valid_mask)(struct gpio_chip *gc,unsigned long *valid_mask,␣
↪unsigned int ngpios);
  int (*add_pin_ranges)(struct gpio_chip *gc);
  int base;
  u16 ngpio;
  const char              *const *names;
  bool can_sleep;
#if IS_ENABLED(CONFIG_GPIO_GENERIC);
  unsigned long (*read_reg)(void __iomem *reg);
  void (*write_reg)(void __iomem *reg, unsigned long data);
  bool be_bits;
  void __iomem *reg_dat;
  void __iomem *reg_set;
  void __iomem *reg_clr;
  void __iomem *reg_dir_out;
  void __iomem *reg_dir_in;
  bool bgpio_dir_unreadable;
  int bgpio_bits;
  spinlock_t bgpio_lock;
  unsigned long bgpio_data;
  unsigned long bgpio_dir;
#endif ;
#ifdef CONFIG_GPIOLIB_IRQCHIP;
  struct gpio_irq_chip irq;
#endif ;
  unsigned long *valid_mask;
#if defined(CONFIG_OF_GPIO);
  struct device_node *of_node;
  unsigned int of_gpio_n_cells;
  int (*of_xlate)(struct gpio_chip *gc, const struct of_phandle_args␣
↪*gpiospec, u32 *flags);
#endif ;
};
```

Members

label a functional name for the GPIO device, such as a part number or the name of the SoC IP-block implementing it.

gpiodev the internal state holder, opaque struct

parent optional parent device providing the GPIOs

owner helps prevent removal of modules exporting active GPIOs

request optional hook for chip-specific activation, such as enabling module power and clock; may sleep

free optional hook for chip-specific deactivation, such as disabling module power and clock; may sleep

get_direction returns direction for signal "offset", 0=out, 1=in, (same as GPIO_LINE_DIRECTION_OUT / GPIO_LINE_DIRECTION_IN), or negative error. It is recommended to always implement this function, even on input-only or output-only gpio chips.

direction_input configures signal "offset" as input, or returns error This can be omitted on input-only or output-only gpio chips.

direction_output configures signal "offset" as output, or returns error This can be omitted on input-only or output-only gpio chips.

get returns value for signal "offset", 0=low, 1=high, or negative error

get_multiple reads values for multiple signals defined by "mask" and stores them in "bits", returns 0 on success or negative error

set assigns output value for signal "offset"

set_multiple assigns output values for multiple signals defined by "mask"

set_config optional hook for all kinds of settings. Uses the same packed config format as generic pinconf.

to_irq optional hook supporting non-static gpio_to_irq() mappings; implementation may not sleep

dbg_show optional routine to show contents in debugfs; default code will be used when this is omitted, but custom code can show extra state (such as pullup/pulldown configuration).

init_valid_mask optional routine to initialize **valid_mask**, to be used if not all GPIOs are valid.

add_pin_ranges optional routine to initialize pin ranges, to be used when requires special mapping of the pins that provides GPIO functionality. It is called after adding GPIO chip and before adding IRQ chip.

base identifies the first GPIO number handled by this chip; or, if negative during registration, requests dynamic ID allocation. DEPRECATION: providing anything non-negative and nailing the base offset of GPIO chips is deprecated. Please pass -1 as base to let gpiolib select the chip base in all possible cases. We want to get rid of the static GPIO number space in the long run.

ngpio the number of GPIOs handled by this controller; the last GPIO handled is (base + ngpio - 1).

names if set, must be an array of strings to use as alternative names for the GPIOs in this chip. Any entry in the array may be NULL if there is no alias for the GPIO, however the array must be **ngpio** entries long. A name can include a single printk format specifier for an unsigned int. It is substituted by the actual number of the gpio.

can_sleep flag must be set iff get()/set() methods sleep, as they must while accessing GPIO expander chips over I2C or SPI. This implies that if the chip supports IRQs, these IRQs need to be threaded as the chip access may sleep when e.g. reading out the IRQ status registers.

read_reg reader function for generic GPIO

write_reg writer function for generic GPIO

be_bits if the generic GPIO has big endian bit order (bit 31 is representing line 0, bit 30 is line 1 ⋯bit 0 is line 31) this is set to true by the generic GPIO core. It is for internal housekeeping only.

reg_dat data (in) register for generic GPIO

reg_set output set register (out=high) for generic GPIO

reg_clr output clear register (out=low) for generic GPIO

reg_dir_out direction out setting register for generic GPIO

reg_dir_in direction in setting register for generic GPIO

bgpio_dir_unreadable indicates that the direction register(s) cannot be read and we need to rely on out internal state tracking.

bgpio_bits number of register bits used for a generic GPIO i.e. <register width> * 8

bgpio_lock used to lock chip->bgpio_data. Also, this is needed to keep shadowed and real data registers writes together.

bgpio_data shadowed data register for generic GPIO to clear/set bits safely.

bgpio_dir shadowed direction register for generic GPIO to clear/set direction safely. A “1” in this word means the line is set as output.

irq Integrates interrupt chip functionality with the GPIO chip. Can be used to handle IRQs for most practical cases.

valid_mask If not `NULL` holds bitmask of GPIOs which are valid to be used from the chip.

of_node Pointer to a device tree node representing this GPIO controller.

of_gpio_n_cells Number of cells used to form the GPIO specifier.

of_xlate Callback to translate a device tree GPIO specifier into a chip- relative GPIO number and flags.

Description

A gpio_chip can help platforms abstract various sources of GPIOs so they can all be accessed through a common programing interface. Example sources would be SOC controllers, FPGAs, multifunction chips, dedicated GPIO expanders, and so on.

Each chip controls a number of signals, identified in method calls by “offset” values in the range 0..(**ngpio** - 1). When those signals are referenced through calls like gpio_get_value(gpio), the offset is calculated by subtracting **base** from the gpio number.

gpiochip_add_data(gc, data)
register a gpio_chip

Parameters

gc undescribed

data driver-private data associated with this chip

Context

potentially before irqs will work

Description

When `gpiochip_add_data()` is called very early during boot, so that GPIOs can be freely used, the chip->parent device must be registered before the gpio framework's arch_initcall(). Otherwise sysfs initialization for GPIOs will fail rudely.

`gpiochip_add_data()` must only be called after gpiolib initialization, ie after core_initcall().

If chip->base is negative, this requests dynamic assignment of a range of valid GPIOs.

Return

A negative errno if the chip can't be registered, such as because the chip->base is invalid or already associated with a different chip. Otherwise it returns zero as a success code.

struct **gpio_pin_range**
pin range controlled by a gpio chip

Definition

```
struct gpio_pin_range {
  struct list_head node;
  struct pinctrl_dev *pctldev;
  struct pinctrl_gpio_range range;
};
```

Members

node list for maintaining set of pin ranges, used internally

pctldev pinctrl device which handles corresponding pins

range actual range of pins controlled by a gpio controller

struct gpio_desc * **gpio_to_desc**(unsigned gpio)
Convert a GPIO number to its descriptor

Parameters

unsigned gpio global GPIO number

Return

The GPIO descriptor associated with the given GPIO, or `NULL` if no GPIO with the given number exists in the system.

struct gpio_desc * **gpiochip_get_desc**(struct gpio_chip * gc, unsigned int hwnum)
get the GPIO descriptor corresponding to the given hardware number for this chip

Parameters

struct gpio_chip * gc GPIO chip

unsigned int hwnum hardware number of the GPIO for this chip

Return

A pointer to the GPIO descriptor or `ERR_PTR(-EINVAL)` if no GPIO exists in the given chip for the specified hardware number.

int **desc_to_gpio**(const struct gpio_desc * desc)
convert a GPIO descriptor to the integer namespace

Parameters

const struct gpio_desc * desc GPIO descriptor

Description

This should disappear in the future but is needed since we still use GPIO numbers for error messages and sysfs nodes.

Return

The global GPIO number for the GPIO specified by its descriptor.

struct gpio_chip * **gpiod_to_chip**(const struct gpio_desc * desc)
Return the GPIO chip to which a GPIO descriptor belongs

Parameters

const struct gpio_desc * desc descriptor to return the chip of

int **gpiod_get_direction**(struct gpio_desc * desc)
return the current direction of a GPIO

Parameters

struct gpio_desc * desc GPIO to get the direction of

Description

Returns 0 for output, 1 for input, or an error code in case of error.

This function may sleep if `gpiod_cansleep()` is true.

void * **gpiochip_get_data**(struct gpio_chip * gc)
get per-subdriver data for the chip

Parameters

struct gpio_chip * gc GPIO chip

Return

The per-subdriver data for the chip.

void **gpiochip_remove**(struct gpio_chip * gc)
unregister a gpio_chip

Parameters

struct gpio_chip * gc the chip to unregister

Description

A gpio_chip with any GPIOs still requested may not be removed.

struct gpio_chip * **gpiochip_find**(void * data, int (*match)(struct gpio_chip *gc, void *data))
iterator for locating a specific gpio_chip

Parameters

void * data data to pass to match function

int (*)(struct gpio_chip *gc, void *data) match Callback function to check gpio_chip

Description

Similar to bus_find_device. It returns a reference to a gpio_chip as determined by a user supplied **match** callback. The callback should return 0 if the device doesn't match and non-zero if it does. If the callback is non-zero, this function will return to the caller and not iterate over any more gpio_chips.

void **gpiochip_set_nested_irqchip**(struct gpio_chip * gc, struct irq_chip * irqchip, unsigned int parent_irq)
connects a nested irqchip to a gpiochip

Parameters

struct gpio_chip * gc the gpiochip to set the irqchip nested handler to

struct irq_chip * irqchip the irqchip to nest to the gpiochip

unsigned int parent_irq the irq number corresponding to the parent IRQ for this nested irqchip

int **gpiochip_irq_map**(struct irq_domain * d, unsigned int irq, irq_hw_number_t hwirq)
maps an IRQ into a GPIO irqchip

Parameters

struct irq_domain * d the irqdomain used by this irqchip

unsigned int irq the global irq number used by this GPIO irqchip irq

irq_hw_number_t hwirq the local IRQ/GPIO line offset on this gpiochip

Description

This function will set up the mapping for a certain IRQ line on a gpiochip by assigning the gpiochip as chip data, and using the irqchip stored inside the gpiochip.

int **gpiochip_irq_domain_activate**(struct irq_domain * domain, struct irq_data * data, bool reserve)
Lock a GPIO to be used as an IRQ

Parameters

struct irq_domain * domain The IRQ domain used by this IRQ chip

struct irq_data * data Outermost irq_data associated with the IRQ

bool reserve If set, only reserve an interrupt vector instead of assigning one

Description

This function is a wrapper that calls `gpiochip_lock_as_irq()` and is to be used as the activate function for the `struct irq_domain_ops`. The host_data for the IRQ domain must be the `struct gpio_chip`.

void **gpiochip_irq_domain_deactivate**(struct irq_domain * domain, struct irq_data * data)

Unlock a GPIO used as an IRQ

Parameters

struct irq_domain * domain The IRQ domain used by this IRQ chip

struct irq_data * data Outermost irq_data associated with the IRQ

Description

This function is a wrapper that will call `gpiochip_unlock_as_irq()` and is to be used as the deactivate function for the `struct irq_domain_ops`. The host_data for the IRQ domain must be the `struct gpio_chip`.

int **gpiochip_irqchip_add_key**(struct gpio_chip * gc, struct irq_chip * irqchip, unsigned int first_irq, irq_flow_handler_t handler, unsigned int type, bool threaded, struct lock_class_key * lock_key, struct lock_class_key * request_key)

adds an irqchip to a gpiochip

Parameters

struct gpio_chip * gc the gpiochip to add the irqchip to

struct irq_chip * irqchip the irqchip to add to the gpiochip

unsigned int first_irq if not dynamically assigned, the base (first) IRQ to allocate gpiochip irqs from

irq_flow_handler_t handler the irq handler to use (often a predefined irq core function)

unsigned int type the default type for IRQs on this irqchip, pass IRQ_TYPE_NONE to have the core avoid setting up any default type in the hardware.

bool threaded whether this irqchip uses a nested thread handler

struct lock_class_key * lock_key lockdep class for IRQ lock

struct lock_class_key * request_key lockdep class for IRQ request

Description

This function closely associates a certain irqchip with a certain gpiochip, providing an irq domain to translate the local IRQs to global irqs in the gpiolib core, and making sure that the gpiochip is passed as chip data to all related functions. Driver callbacks need to use `gpiochip_get_data()` to get their local state containers

back from the gpiochip passed as chip data. An irqdomain will be stored in the gpiochip that shall be used by the driver to handle IRQ number translation. The gpiochip will need to be initialized and registered before calling this function.

This function will handle two cell:ed simple IRQs and assumes all the pins on the gpiochip can generate a unique IRQ. Everything else need to be open coded.

int **gpiochip_irqchip_add_domain**(struct gpio_chip * gc, struct irq_domain * domain)
adds an irqdomain to a gpiochip

Parameters

struct gpio_chip * gc the gpiochip to add the irqchip to

struct irq_domain * domain the irqdomain to add to the gpiochip

Description

This function adds an IRQ domain to the gpiochip.

int **gpiochip_generic_request**(struct gpio_chip * gc, unsigned offset)
request the gpio function for a pin

Parameters

struct gpio_chip * gc the gpiochip owning the GPIO

unsigned offset the offset of the GPIO to request for GPIO function

void **gpiochip_generic_free**(struct gpio_chip * gc, unsigned offset)
free the gpio function from a pin

Parameters

struct gpio_chip * gc the gpiochip to request the gpio function for

unsigned offset the offset of the GPIO to free from GPIO function

int **gpiochip_generic_config**(struct gpio_chip * gc, unsigned offset, unsigned long config)
apply configuration for a pin

Parameters

struct gpio_chip * gc the gpiochip owning the GPIO

unsigned offset the offset of the GPIO to apply the configuration

unsigned long config the configuration to be applied

int **gpiochip_add_pingroup_range**(struct gpio_chip * gc, struct pinctrl_dev * pctldev, unsigned int gpio_offset, const char * pin_group)
add a range for GPIO <-> pin mapping

Parameters

struct gpio_chip * gc the gpiochip to add the range for

struct pinctrl_dev * pctldev the pin controller to map to

unsigned int gpio_offset the start offset in the current gpio_chip number space

const char * pin_group name of the pin group inside the pin controller

Description

Calling this function directly from a DeviceTree-supported pinctrl driver is DEPRECATED. Please see Section 2.1 of Documentation/devicetree/bindings/gpio/gpio.txt on how to bind pinctrl and gpio drivers via the "gpio-ranges" property.

int **gpiochip_add_pin_range**(struct gpio_chip * gc, const char * pinctl_name, unsigned int gpio_offset, unsigned int pin_offset, unsigned int npins)
add a range for GPIO <-> pin mapping

Parameters

struct gpio_chip * gc the gpiochip to add the range for

const char * pinctl_name the dev_name() of the pin controller to map to

unsigned int gpio_offset the start offset in the current gpio_chip number space

unsigned int pin_offset the start offset in the pin controller number space

unsigned int npins the number of pins from the offset of each pin space (GPIO and pin controller) to accumulate in this range

Return

0 on success, or a negative error-code on failure.

Description

Calling this function directly from a DeviceTree-supported pinctrl driver is DEPRECATED. Please see Section 2.1 of Documentation/devicetree/bindings/gpio/gpio.txt on how to bind pinctrl and gpio drivers via the "gpio-ranges" property.

void **gpiochip_remove_pin_ranges**(struct gpio_chip * gc)
remove all the GPIO <-> pin mappings

Parameters

struct gpio_chip * gc the chip to remove all the mappings for

const char * **gpiochip_is_requested**(struct gpio_chip * gc, unsigned offset)
return string iff signal was requested

Parameters

struct gpio_chip * gc controller managing the signal

unsigned offset of signal within controller's 0..(ngpio - 1) range

Description

Returns NULL if the GPIO is not currently requested, else a string. The string returned is the label passed to gpio_request(); if none has been passed it is a meaningless, non-NULL constant.

This function is for use by GPIO controller drivers. The label can help with diagnostics, and knowing that the signal is used as a GPIO can help avoid accidentally multiplexing it to another controller.

struct gpio_desc * **gpiochip_request_own_desc**(struct gpio_chip * gc, unsigned int hwnum, const char * label, enum gpio_lookup_flags lflags, enum gpiod_flags dflags)
Allow GPIO chip to request its own descriptor

Parameters

struct gpio_chip * gc GPIO chip

unsigned int hwnum hardware number of the GPIO for which to request the descriptor

const char * label label for the GPIO

enum gpio_lookup_flags lflags lookup flags for this GPIO or 0 if default, this can be used to specify things like line inversion semantics with the machine flags such as GPIO_OUT_LOW

enum gpiod_flags dflags descriptor request flags for this GPIO or 0 if default, this can be used to specify consumer semantics such as open drain

Description

Function allows GPIO chip drivers to request and use their own GPIO descriptors via gpiolib API. Difference to gpiod_request() is that this function will not increase reference count of the GPIO chip module. This allows the GPIO chip module to be unloaded as needed (we assume that the GPIO chip driver handles freeing the GPIOs it has requested).

Return

A pointer to the GPIO descriptor, or an ERR_PTR()-encoded negative error code on failure.

void **gpiochip_free_own_desc**(struct gpio_desc * desc)
Free GPIO requested by the chip driver

Parameters

struct gpio_desc * desc GPIO descriptor to free

Description

Function frees the given GPIO requested previously with `gpiochip_request_own_desc()`.

int **gpiod_direction_input**(struct gpio_desc * desc)
set the GPIO direction to input

Parameters

struct gpio_desc * desc GPIO to set to input

Description

Set the direction of the passed GPIO to input, such as `gpiod_get_value()` can be called safely on it.

Return 0 in case of success, else an error code.

int **gpiod_direction_output_raw**(struct gpio_desc * desc, int value)
set the GPIO direction to output

Parameters

struct gpio_desc * desc GPIO to set to output

int value initial output value of the GPIO

Description

Set the direction of the passed GPIO to output, such as `gpiod_set_value()` can be called safely on it. The initial value of the output must be specified as raw value on the physical line without regard for the ACTIVE_LOW status.

Return 0 in case of success, else an error code.

int **gpiod_direction_output**(struct gpio_desc * desc, int value)
set the GPIO direction to output

Parameters

struct gpio_desc * desc GPIO to set to output

int value initial output value of the GPIO

Description

Set the direction of the passed GPIO to output, such as `gpiod_set_value()` can be called safely on it. The initial value of the output must be specified as the logical value of the GPIO, i.e. taking its ACTIVE_LOW status into account.

Return 0 in case of success, else an error code.

int **gpiod_set_config**(struct gpio_desc * desc, unsigned long config)
sets **config** for a GPIO

Parameters

struct gpio_desc * desc descriptor of the GPIO for which to set the configuration

unsigned long config Same packed config format as generic pinconf

Return

0 on success, `-ENOTSUPP` if the controller doesn't support setting the configuration.

int **gpiod_set_debounce**(struct gpio_desc * desc, unsigned debounce)
sets **debounce** time for a GPIO

Parameters

struct gpio_desc * desc descriptor of the GPIO for which to set debounce time

unsigned debounce debounce time in microseconds

Return

0 on success, `-ENOTSUPP` if the controller doesn't support setting the debounce time.

int **gpiod_set_transitory**(struct gpio_desc * desc, bool transitory)
Lose or retain GPIO state on suspend or reset

Parameters

struct gpio_desc * desc descriptor of the GPIO for which to configure persistence

bool transitory True to lose state on suspend or reset, false for persistence

Return

0 on success, otherwise a negative error code.

int **gpiod_is_active_low**(const struct gpio_desc * desc)
test whether a GPIO is active-low or not

Parameters

const struct gpio_desc * desc the gpio descriptor to test

Description

Returns 1 if the GPIO is active-low, 0 otherwise.

void **gpiod_toggle_active_low**(struct gpio_desc * desc)
toggle whether a GPIO is active-low or not

Parameters

struct gpio_desc * desc the gpio descriptor to change

int **gpiod_get_raw_value**(const struct gpio_desc * desc)
return a gpio's raw value

Parameters

const struct gpio_desc * desc gpio whose value will be returned

Description

Return the GPIO's raw value, i.e. the value of the physical line disregarding its ACTIVE_LOW status, or negative errno on failure.

This function can be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

int **gpiod_get_value**(const struct gpio_desc * desc)
return a gpio's value

Parameters

const struct gpio_desc * desc gpio whose value will be returned

Description

Return the GPIO's logical value, i.e. taking the ACTIVE_LOW status into account, or negative errno on failure.

This function can be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

int **gpiod_get_raw_array_value**(unsigned int array_size, struct gpio_desc ** desc_array, struct gpio_array * array_info, unsigned long * value_bitmap)
read raw values from an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor array / value bitmap

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be read

struct gpio_array * array_info information on applicability of fast bitmap processing path

unsigned long * value_bitmap bitmap to store the read values

Description

Read the raw values of the GPIOs, i.e. the values of the physical lines without regard for their ACTIVE_LOW status. Return 0 in case of success, else an error code.

This function can be called from contexts where we cannot sleep, and it will complain if the GPIO chip functions potentially sleep.

int **gpiod_get_array_value**(unsigned int array_size, struct gpio_desc ** desc_array, struct gpio_array * array_info, unsigned long * value_bitmap)
read values from an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor array / value bitmap

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be read

struct gpio_array * array_info information on applicability of fast bitmap processing path

unsigned long * value_bitmap bitmap to store the read values

Description

Read the logical values of the GPIOs, i.e. taking their ACTIVE_LOW status into account. Return 0 in case of success, else an error code.

This function can be called from contexts where we cannot sleep, and it will complain if the GPIO chip functions potentially sleep.

void **gpiod_set_raw_value**(struct gpio_desc * desc, int value)
assign a gpio’s raw value

Parameters

struct gpio_desc * desc gpio whose value will be assigned

int value value to assign

Description

Set the raw value of the GPIO, i.e. the value of its physical line without regard for its ACTIVE_LOW status.

This function can be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

void **gpiod_set_value**(struct gpio_desc * desc, int value)
assign a gpio's value

Parameters

struct gpio_desc * desc gpio whose value will be assigned

int value value to assign

Description

Set the logical value of the GPIO, i.e. taking its ACTIVE_LOW, OPEN_DRAIN and OPEN_SOURCE flags into account.

This function can be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

int **gpiod_set_raw_array_value**(unsigned int array_size, struct gpio_desc ** desc_array, struct gpio_array * array_info, unsigned long * value_bitmap)
assign values to an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor array / value bitmap

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be assigned

struct gpio_array * array_info information on applicability of fast bitmap processing path

unsigned long * value_bitmap bitmap of values to assign

Description

Set the raw values of the GPIOs, i.e. the values of the physical lines without regard for their ACTIVE_LOW status.

This function can be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

int **gpiod_set_array_value**(unsigned int array_size, struct gpio_desc ** desc_array, struct gpio_array * array_info, unsigned long * value_bitmap)
assign values to an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor array / value bitmap

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be assigned

struct gpio_array * array_info information on applicability of fast bitmap processing path

unsigned long * value_bitmap bitmap of values to assign

Description

Set the logical values of the GPIOs, i.e. taking their ACTIVE_LOW status into account.

This function can be called from contexts where we cannot sleep, and will complain if the GPIO chip functions potentially sleep.

int **gpiod_cansleep**(const struct gpio_desc * desc)
report whether gpio value access may sleep

Parameters

const struct gpio_desc * desc gpio to check

int **gpiod_set_consumer_name**(struct gpio_desc * desc, const char * name)
set the consumer name for the descriptor

Parameters

struct gpio_desc * desc gpio to set the consumer name on

const char * name the new consumer name

int **gpiod_to_irq**(const struct gpio_desc * desc)
return the IRQ corresponding to a GPIO

Parameters

const struct gpio_desc * desc gpio whose IRQ will be returned (already requested)

Description

Return the IRQ corresponding to the passed GPIO, or an error code in case of error.

int **gpiochip_lock_as_irq**(struct gpio_chip * gc, unsigned int offset)
lock a GPIO to be used as IRQ

Parameters

struct gpio_chip * gc the chip the GPIO to lock belongs to

unsigned int offset the offset of the GPIO to lock as IRQ

Description

This is used directly by GPIO drivers that want to lock down a certain GPIO line to be used for IRQs.

void **gpiochip_unlock_as_irq**(struct gpio_chip * gc, unsigned int offset)
unlock a GPIO used as IRQ

Parameters

struct gpio_chip * gc the chip the GPIO to lock belongs to

unsigned int offset the offset of the GPIO to lock as IRQ

Description

This is used directly by GPIO drivers that want to indicate that a certain GPIO is no longer used exclusively for IRQ.

int **gpiod_get_raw_value_cansleep**(const struct gpio_desc * desc)
return a gpio's raw value

Parameters

const struct gpio_desc * desc gpio whose value will be returned

Description

Return the GPIO's raw value, i.e. the value of the physical line disregarding its ACTIVE_LOW status, or negative errno on failure.

This function is to be called from contexts that can sleep.

int **gpiod_get_value_cansleep**(const struct gpio_desc * desc)
return a gpio's value

Parameters

const struct gpio_desc * desc gpio whose value will be returned

Description

Return the GPIO's logical value, i.e. taking the ACTIVE_LOW status into account, or negative errno on failure.

This function is to be called from contexts that can sleep.

int **gpiod_get_raw_array_value_cansleep**(unsigned int array_size, struct gpio_desc ** desc_array, struct gpio_array * array_info, unsigned long * value_bitmap)
read raw values from an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor array / value bitmap

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be read

struct gpio_array * array_info information on applicability of fast bitmap processing path

unsigned long * value_bitmap bitmap to store the read values

Description

Read the raw values of the GPIOs, i.e. the values of the physical lines without regard for their ACTIVE_LOW status. Return 0 in case of success, else an error code.

This function is to be called from contexts that can sleep.

int **gpiod_get_array_value_cansleep**(unsigned int array_size, struct gpio_desc ** desc_array, struct gpio_array * array_info, unsigned long * value_bitmap)
read values from an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor array / value bitmap

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be read

struct gpio_array * array_info information on applicability of fast bitmap processing path

unsigned long * value_bitmap bitmap to store the read values

Description

Read the logical values of the GPIOs, i.e. taking their ACTIVE_LOW status into account. Return 0 in case of success, else an error code.

This function is to be called from contexts that can sleep.

void **gpiod_set_raw_value_cansleep**(struct gpio_desc * desc, int value)
assign a gpio's raw value

Parameters

struct gpio_desc * desc gpio whose value will be assigned

int value value to assign

Description

Set the raw value of the GPIO, i.e. the value of its physical line without regard for its ACTIVE_LOW status.

This function is to be called from contexts that can sleep.

void **gpiod_set_value_cansleep**(struct gpio_desc * desc, int value)
assign a gpio's value

Parameters

struct gpio_desc * desc gpio whose value will be assigned

int value value to assign

Description

Set the logical value of the GPIO, i.e. taking its ACTIVE_LOW status into account

This function is to be called from contexts that can sleep.

int **gpiod_set_raw_array_value_cansleep**(unsigned int array_size, struct gpio_desc ** desc_array, struct gpio_array * array_info, unsigned long * value_bitmap)

assign values to an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor array / value bitmap

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be assigned

struct gpio_array * array_info information on applicability of fast bitmap processing path

unsigned long * value_bitmap bitmap of values to assign

Description

Set the raw values of the GPIOs, i.e. the values of the physical lines without regard for their ACTIVE_LOW status.

This function is to be called from contexts that can sleep.

int **gpiod_set_array_value_cansleep**(unsigned int array_size, struct gpio_desc ** desc_array, struct gpio_array * array_info, unsigned long * value_bitmap)
assign values to an array of GPIOs

Parameters

unsigned int array_size number of elements in the descriptor array / value bitmap

struct gpio_desc ** desc_array array of GPIO descriptors whose values will be assigned

struct gpio_array * array_info information on applicability of fast bitmap processing path

unsigned long * value_bitmap bitmap of values to assign

Description

Set the logical values of the GPIOs, i.e. taking their ACTIVE_LOW status into account.

This function is to be called from contexts that can sleep.

void **gpiod_add_lookup_table**(struct gpiod_lookup_table * table)
register GPIO device consumers

Parameters

struct gpiod_lookup_table * table table of consumers to register

void **gpiod_remove_lookup_table**(struct gpiod_lookup_table * table)
unregister GPIO device consumers

Parameters

struct gpiod_lookup_table * table table of consumers to unregister

void **gpiod_add_hogs**(struct gpiod_hog * hogs)
register a set of GPIO hogs from machine code

Parameters

struct gpiod_hog * hogs table of gpio hog entries with a zeroed sentinel at the end

struct gpio_desc * **fwnode_gpiod_get_index**(struct fwnode_handle * fwnode, const char * con_id, int index, enum gpiod_flags flags, const char * label)
obtain a GPIO from firmware node

Parameters

struct fwnode_handle * fwnode handle of the firmware node

const char * con_id function within the GPIO consumer

int index index of the GPIO to obtain for the consumer

enum gpiod_flags flags GPIO initialization flags

const char * label label to attach to the requested GPIO

Description

This function can be used for drivers that get their configuration from opaque firmware.

The function properly finds the corresponding GPIO using whatever is the underlying firmware interface and then makes sure that the GPIO descriptor is requested before it is returned to the caller.

In case of error an ERR_PTR() is returned.

Return

On successful request the GPIO pin is configured in accordance with provided **flags**.

int **gpiod_count**(struct device * dev, const char * con_id)
return the number of GPIOs associated with a device / function or -ENOENT if no GPIO has been assigned to the requested function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

struct gpio_desc * **gpiod_get**(struct device * dev, const char * con_id, enum gpiod_flags flags)
obtain a GPIO for a given GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Return the GPIO descriptor corresponding to the function con_id of device dev, -ENOENT if no GPIO has been assigned to the requested function, or another IS_ERR() code if an error occurred while trying to acquire the GPIO.

struct gpio_desc * **gpiod_get_optional**(struct device * dev, const char * con_id, enum gpiod_flags flags)
obtain an optional GPIO for a given GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

This is equivalent to `gpiod_get()`, except that when no GPIO was assigned to the requested function it will return NULL. This is convenient for drivers that need to handle optional GPIOs.

struct gpio_desc * **gpiod_get_index**(struct device * dev, const char * con_id, unsigned int idx, enum gpiod_flags flags)
obtain a GPIO from a multi-index GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

unsigned int idx index of the GPIO to obtain in the consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

This variant of `gpiod_get()` allows to access GPIOs other than the first defined one for functions that define several GPIOs.

Return a valid GPIO descriptor, -ENOENT if no GPIO has been assigned to the requested function and/or index, or another IS_ERR() code if an error occurred while trying to acquire the GPIO.

struct gpio_desc * **fwnode_get_named_gpiod**(struct fwnode_handle * fwnode, const char * propname, int index, enum gpiod_flags dflags, const char * label)
obtain a GPIO from firmware node

Parameters

struct fwnode_handle * fwnode handle of the firmware node

const char * propname name of the firmware property representing the GPIO

int index index of the GPIO to obtain for the consumer

enum gpiod_flags dflags GPIO initialization flags

const char * label label to attach to the requested GPIO

Description

This function can be used for drivers that get their configuration from opaque firmware.

The function properly finds the corresponding GPIO using whatever is the underlying firmware interface and then makes sure that the GPIO descriptor is requested before it is returned to the caller.

In case of error an ERR_PTR() is returned.

Return

On successful request the GPIO pin is configured in accordance with provided **dflags**.

struct gpio_desc * **gpiod_get_index_optional**(struct device * dev, const char * con_id, unsigned int index, enum gpiod_flags flags)

obtain an optional GPIO from a multi-index GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

unsigned int index index of the GPIO to obtain in the consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

This is equivalent to `gpiod_get_index()`, except that when no GPIO with the specified index was assigned to the requested function it will return NULL. This is convenient for drivers that need to handle optional GPIOs.

struct gpio_descs * **gpiod_get_array**(struct device * dev, const char * con_id, enum gpiod_flags flags)

obtain multiple GPIOs from a multi-index GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

This function acquires all the GPIOs defined under a given function.

Return a struct gpio_descs containing an array of descriptors, -ENOENT if no GPIO has been assigned to the requested function, or another IS_ERR() code if an error occurred while trying to acquire the GPIOs.

struct gpio_descs * **gpiod_get_array_optional**(struct device * dev, const char * con_id, enum gpiod_flags flags)
obtain multiple GPIOs from a multi-index GPIO function

Parameters

struct device * dev GPIO consumer, can be NULL for system-global GPIOs

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

This is equivalent to `gpiod_get_array()`, except that when no GPIO was assigned to the requested function it will return NULL.

void **gpiod_put**(struct gpio_desc * desc)
dispose of a GPIO descriptor

Parameters

struct gpio_desc * desc GPIO descriptor to dispose of

Description

No descriptor can be used after `gpiod_put()` has been called on it.

void **gpiod_put_array**(struct gpio_descs * descs)
dispose of multiple GPIO descriptors

Parameters

struct gpio_descs * descs struct gpio_descs containing an array of descriptors

51.10 ACPI support

void **acpi_gpiochip_request_interrupts**(struct gpio_chip * chip)
Register isr for gpio chip ACPI events

Parameters

struct gpio_chip * chip GPIO chip

Description

ACPI5 platforms can use GPIO signaled ACPI events. These GPIO interrupts are handled by ACPI event methods which need to be called from the GPIO chip's interrupt handler. `acpi_gpiochip_request_interrupts()` finds out which GPIO pins have ACPI event methods and assigns interrupt handlers that calls the ACPI event methods for those pins.

void **acpi_gpiochip_free_interrupts**(struct gpio_chip * chip)
Free GPIO ACPI event interrupts.

Parameters

struct gpio_chip * chip GPIO chip

Description

Free interrupts associated with GPIO ACPI event method for the given GPIO chip.

int **acpi_dev_gpio_irq_get**(struct acpi_device * adev, int index)
Find GpioInt and translate it to Linux IRQ number

Parameters

struct acpi_device * adev pointer to a ACPI device to get IRQ from

int index index of GpioInt resource (starting from 0)

Description

If the device has one or more GpioInt resources, this function can be used to translate from the GPIO offset in the resource to the Linux IRQ number.

The function is idempotent, though each time it runs it will configure GPIO pin direction according to the flags in GpioInt resource.

Return

Linux IRQ number (> 0) on success, negative errno on failure.

51.11 Device tree support

struct gpio_desc * **gpiod_get_from_of_node**(struct device_node * node, const char * propname, int index, enum gpiod_flags dflags, const char * label)
obtain a GPIO from an OF node

Parameters

struct device_node * node handle of the OF node

const char * propname name of the DT property representing the GPIO

int index index of the GPIO to obtain for the consumer

enum gpiod_flags dflags GPIO initialization flags

const char * label label to attach to the requested GPIO

Return

On successful request the GPIO pin is configured in accordance with provided **dflags**.

Description

In case of error an ERR_PTR() is returned.

int **of_mm_gpiochip_add_data**(struct device_node * np, struct of_mm_gpio_chip * mm_gc, void * data)
Add memory mapped GPIO chip (bank)

Parameters

struct device_node * np device node of the GPIO chip

struct of_mm_gpio_chip * mm_gc pointer to the of_mm_gpio_chip allocated structure

void * data driver data to store in the struct gpio_chip

Description

To use this function you should allocate and fill mm_gc with:

1) In the gpio_chip structure: - all the callbacks - of_gpio_n_cells - of_xlate callback (optional)

3) In the of_mm_gpio_chip structure: - save_regs callback (optional)

If succeeded, this function will map bank's memory and will do all necessary work for you. Then you'll able to use .regs to manage GPIOs from the callbacks.

void **of_mm_gpiochip_remove**(struct of_mm_gpio_chip * mm_gc)
Remove memory mapped GPIO chip (bank)

Parameters

struct of_mm_gpio_chip * mm_gc pointer to the of_mm_gpio_chip allocated structure

51.12 Device-managed API

struct gpio_desc * **devm_gpiod_get**(struct device * dev, const char * con_id, enum gpiod_flags flags)
Resource-managed `gpiod_get()`

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed `gpiod_get()`. GPIO descriptors returned from this function are automatically disposed on driver detach. See `gpiod_get()` for detailed information about behavior and return values.

struct gpio_desc * **devm_gpiod_get_optional**(struct device * dev, const char * con_id, enum gpiod_flags flags)
Resource-managed `gpiod_get_optional()`

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed `gpiod_get_optional()`. GPIO descriptors returned from this function are automatically disposed on driver detach. See `gpiod_get_optional()` for detailed information about behavior and return values.

struct gpio_desc * **devm_gpiod_get_index**(struct device * dev, const char * con_id, unsigned int idx, enum gpiod_flags flags)

Resource-managed `gpiod_get_index()`

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

unsigned int idx index of the GPIO to obtain in the consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed `gpiod_get_index()`. GPIO descriptors returned from this function are automatically disposed on driver detach. See `gpiod_get_index()` for detailed information about behavior and return values.

struct gpio_desc * **devm_gpiod_get_from_of_node**(struct device * dev, struct device_node * node, const char * propname, int index, enum gpiod_flags dflags, const char * label)

obtain a GPIO from an OF node

Parameters

struct device * dev device for lifecycle management

struct device_node * node handle of the OF node

const char * propname name of the DT property representing the GPIO

int index index of the GPIO to obtain for the consumer

enum gpiod_flags dflags GPIO initialization flags

const char * label label to attach to the requested GPIO

Return

On successful request the GPIO pin is configured in accordance with provided **dflags**.

Description

In case of error an ERR_PTR() is returned.

struct gpio_desc * **devm_fwnode_gpiod_get_index**(struct device * dev, struct fwnode_handle * fwnode, const char * con_id, int index, enum gpiod_flags flags, const char * label)

get a GPIO descriptor from a given node

Parameters

struct device * dev GPIO consumer

struct fwnode_handle * fwnode firmware node containing GPIO reference

const char * con_id function within the GPIO consumer

int index index of the GPIO to obtain in the consumer

enum gpiod_flags flags GPIO initialization flags

const char * label label to attach to the requested GPIO

Description

GPIO descriptors returned from this function are automatically disposed on driver detach.

On successful request the GPIO pin is configured in accordance with provided **flags**.

struct gpio_desc * **devm_gpiod_get_index_optional**(struct device * dev, const char * con_id, unsigned int index, enum gpiod_flags flags)

Resource-managed `gpiod_get_index_optional()`

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

unsigned int index index of the GPIO to obtain in the consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed `gpiod_get_index_optional()`. GPIO descriptors returned from this function are automatically disposed on driver detach. See `gpiod_get_index_optional()` for detailed information about behavior and return values.

struct gpio_descs * **devm_gpiod_get_array**(struct device * dev, const char * con_id, enum gpiod_flags flags)

Resource-managed `gpiod_get_array()`

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed `gpiod_get_array()`. GPIO descriptors returned from this function are automatically disposed on driver detach. See `gpiod_get_array()` for detailed information about behavior and return values.

struct gpio_descs * **devm_gpiod_get_array_optional**(struct device * dev, const char * con_id, enum gpiod_flags flags)
Resource-managed `gpiod_get_array_optional()`

Parameters

struct device * dev GPIO consumer

const char * con_id function within the GPIO consumer

enum gpiod_flags flags optional GPIO initialization flags

Description

Managed `gpiod_get_array_optional()`. GPIO descriptors returned from this function are automatically disposed on driver detach. See `gpiod_get_array_optional()` for detailed information about behavior and return values.

void **devm_gpiod_put**(struct device * dev, struct gpio_desc * desc)
Resource-managed `gpiod_put()`

Parameters

struct device * dev GPIO consumer

struct gpio_desc * desc GPIO descriptor to dispose of

Description

Dispose of a GPIO descriptor obtained with `devm_gpiod_get()` or `devm_gpiod_get_index()`. Normally this function will not be called as the GPIO will be disposed of by the resource management code.

void **devm_gpiod_unhinge**(struct device * dev, struct gpio_desc * desc)
Remove resource management from a gpio descriptor

Parameters

struct device * dev GPIO consumer

struct gpio_desc * desc GPIO descriptor to remove resource management from

Description

Remove resource management from a GPIO descriptor. This is needed when you want to hand over lifecycle management of a descriptor to another mechanism.

void **devm_gpiod_put_array**(struct device * dev, struct gpio_descs * descs)
Resource-managed `gpiod_put_array()`

Parameters

struct device * dev GPIO consumer

struct gpio_descs * descs GPIO descriptor array to dispose of

Description

Dispose of an array of GPIO descriptors obtained with `devm_gpiod_get_array()`. Normally this function will not be called as the GPIOs will be disposed of by the resource management code.

int **devm_gpio_request**(struct device * dev, unsigned gpio, const char * label)
request a GPIO for a managed device

Parameters

struct device * dev device to request the GPIO for

unsigned gpio GPIO to allocate

const char * label the name of the requested GPIO

Except for the extra **dev** argument, this function takes the same arguments and performs the same function as gpio_request(). GPIOs requested with this function will be automatically freed on driver detach.

If an GPIO allocated with this function needs to be freed separately, `devm_gpio_free()` must be used.

int **devm_gpio_request_one**(struct device * dev, unsigned gpio, unsigned long flags, const char * label)
request a single GPIO with initial setup

Parameters

struct device * dev device to request for

unsigned gpio the GPIO number

unsigned long flags GPIO configuration as specified by GPIOF_*

const char * label a literal description string of this GPIO

void **devm_gpio_free**(struct device * dev, unsigned int gpio)
free a GPIO

Parameters

struct device * dev device to free GPIO for

unsigned int gpio GPIO to free

Except for the extra **dev** argument, this function takes the same arguments and performs the same function as gpio_free(). This function instead of gpio_free() should be used to manually free GPIOs allocated with `devm_gpio_request()`.

int **devm_gpiochip_add_data**(struct device * dev, struct gpio_chip * gc, void * data)
Resource managed `gpiochip_add_data()`

Parameters

struct device * dev pointer to the device that gpio_chip belongs to.

struct gpio_chip * gc the GPIO chip to register

void * data driver-private data associated with this chip

Context

potentially before irqs will work

Description

The gpio chip automatically be released when the device is unbound.

Return

A negative errno if the chip can't be registered, such as because the gc->base is invalid or already associated with a different chip. Otherwise it returns zero as a success code.

51.13 sysfs helpers

int **gpiod_export**(struct gpio_desc * desc, bool direction_may_change)
export a GPIO through sysfs

Parameters

struct gpio_desc * desc GPIO to make available, already requested

bool direction_may_change true if userspace may change GPIO direction

Context

arch_initcall or later

Description

When drivers want to make a GPIO accessible to userspace after they have requested it – perhaps while debugging, or as part of their public interface – they may use this routine. If the GPIO can change direction (some can't) and the caller allows it, userspace will see "direction" sysfs attribute which may be used to change the gpio's direction. A "value" attribute will always be provided.

Returns zero on success, else an error.

int **gpiod_export_link**(struct device * dev, const char * name, struct gpio_desc * desc)
create a sysfs link to an exported GPIO node

Parameters

struct device * dev device under which to create symlink

const char * name name of the symlink

struct gpio_desc * desc GPIO to create symlink to, already exported

Description

Set up a symlink from /sys/⋯/dev/name to /sys/class/gpio/gpioN node. Caller is responsible for unlinking.

Returns zero on success, else an error.

void **gpiod_unexport**(struct gpio_desc * desc)
reverse effect of `gpiod_export()`

Parameters

struct gpio_desc * desc GPIO to make unavailable

Description

This is implicit on gpiod_free().

CHAPTER

FIFTYTWO

RAID

52.1 MD Cluster

The cluster MD is a shared-device RAID for a cluster, it supports two levels: raid1 and raid10 (limited support).

52.1.1 1. On-disk format

Separate write-intent-bitmaps are used for each cluster node. The bitmaps record all writes that may have been started on that node, and may not yet have finished. The on-disk layout is:

```
0                    4k                     8k                    12k
-------------------------------------------------------------------
| idle                | md super            | bm super [0] + bits |
| bm bits[0, contd]   | bm super[1] + bits  | bm bits[1, contd]   |
| bm super[2] + bits  | bm bits [2, contd]  | bm super[3] + bits  |
| bm bits [3, contd]  |                     |                     |
```

During "normal" functioning we assume the filesystem ensures that only one node writes to any given block at a time, so a write request will

- set the appropriate bit (if not already set)
- commit the write to all mirrors
- schedule the bit to be cleared after a timeout.

Reads are just handled normally. It is up to the filesystem to ensure one node doesn't read from a location where another node (or the same node) is writing.

52.1.2 2. DLM Locks for management

There are three groups of locks for managing the device:

2.1 Bitmap lock resource (bm_lockres)

The bm_lockres protects individual node bitmaps. They are named in the form bitmap000 for node 1, bitmap001 for node 2 and so on. When a node joins the cluster, it acquires the lock in PW mode and it stays so during the lifetime the node is part of the cluster. The lock resource number is based on the slot number returned by the DLM subsystem. Since DLM starts node count from one and bitmap slots start from zero, one is subtracted from the DLM slot number to arrive at the bitmap slot number.

The LVB of the bitmap lock for a particular node records the range of sectors that are being re-synced by that node. No other node may write to those sectors. This is used when a new nodes joins the cluster.

2.2 Message passing locks

Each node has to communicate with other nodes when starting or ending resync, and for metadata superblock updates. This communication is managed through three locks: “token”, “message”, and “ack”, together with the Lock Value Block (LVB) of one of the “message” lock.

2.3 new-device management

A single lock: “no-new-dev” is used to co-ordinate the addition of new devices - this must be synchronized across the array. Normally all nodes hold a concurrent-read lock on this device.

52.1.3 3. Communication

Messages can be broadcast to all nodes, and the sender waits for all other nodes to acknowledge the message before proceeding. Only one message can be processed at a time.

3.1 Message Types

There are six types of messages which are passed:

3.1.1 METADATA_UPDATED

informs other nodes that the metadata has been updated, and the node must re-read the md superblock. This is performed synchronously. It is primarily used to signal device failure.

3.1.2 RESYNCING

informs other nodes that a resync is initiated or ended so that each node may suspend or resume the region. Each RESYNCING message identifies a range of the devices that the sending node is about to resync. This overrides any previous notification from that node: only one ranged can be resynced at a time per-node.

3.1.3 NEWDISK

informs other nodes that a device is being added to the array. Message contains an identifier for that device. See below for further details.

3.1.4 REMOVE

A failed or spare device is being removed from the array. The slot-number of the device is included in the message.

3.1.5 RE_ADD:

A failed device is being re-activated - the assumption is that it has been determined to be working again.

3.1.6 BITMAP_NEEDS_SYNC:

If a node is stopped locally but the bitmap isn't clean, then another node is informed to take the ownership of resync.

3.2 Communication mechanism

The DLM LVB is used to communicate within nodes of the cluster. There are three resources used for the purpose:

3.2.1 token

The resource which protects the entire communication system. The node having the token resource is allowed to communicate.

3.2.2 message

The lock resource which carries the data to communicate.

3.2.3 ack

The resource, acquiring which means the message has been acknowledged by all nodes in the cluster. The BAST of the resource is used to inform the receiving node that a node wants to communicate.

The algorithm is:

1. receive status - all nodes have concurrent-reader lock on “ack”:

```
sender                          receiver                receiver
"ack":CR                        "ack":CR                "ack":CR
```

2. sender get EX on “token”, sender get EX on “message”:

```
sender                          receiver                receiver
"token":EX                      "ack":CR                "ack":CR
"message":EX
"ack":CR
```

Sender checks that it still needs to send a message. Messages received or other events that happened while waiting for the “token” may have made this message inappropriate or redundant.

3. sender writes LVB

sender down-convert “message” from EX to CW

sender try to get EX of “ack”

```
 [ wait until all receivers have *processed* the "message" ]

                                  [ triggered by bast of "ack" ]
                                  receiver get CR on "message"
                                  receiver read LVB
                                  receiver processes the message
                                  [ wait finish ]
                                  receiver releases "ack"
                                  receiver tries to get PR on "message
↪"

sender                          receiver                receiver
"token":EX                      "message":CR            "message":CR
"message":CW
"ack":EX
```

4. triggered by grant of EX on “ack” (indicating all receivers have processed message)

sender down-converts “ack” from EX to CR

sender releases “message”

sender releases “token”

```
                            receiver upconvert to PR on "message"
                            receiver get CR of "ack"
                            receiver release "message"
```

(continues on next page)

(continued from previous page)

```
sender                          receiver                        receiver
"ack":CR                        "ack":CR                        "ack":CR
```

52.1.4 4. Handling Failures

4.1 Node Failure

When a node fails, the DLM informs the cluster with the slot number. The node starts a cluster recovery thread. The cluster recovery thread:

- acquires the bitmap<number> lock of the failed node
- opens the bitmap
- reads the bitmap of the failed node
- copies the set bitmap to local node
- cleans the bitmap of the failed node
- releases bitmap<number> lock of the failed node
- initiates resync of the bitmap on the current node md_check_recovery is invoked within recover_bitmaps, then md_check_recovery -> metadata_update_start/finish, it will lock the communication by lock_comm. Which means when one node is resyncing it blocks all other nodes from writing anywhere on the array.

The resync process is the regular md resync. However, in a clustered environment when a resync is performed, it needs to tell other nodes of the areas which are suspended. Before a resync starts, the node send out RESYNCING with the (lo,hi) range of the area which needs to be suspended. Each node maintains a suspend_list, which contains the list of ranges which are currently suspended. On receiving RESYNCING, the node adds the range to the suspend_list. Similarly, when the node performing resync finishes, it sends RESYNCING with an empty range to other nodes and other nodes remove the corresponding entry from the suspend_list.

A helper function, ->area_resyncing() can be used to check if a particular I/O range should be suspended or not.

52.1.5 4.2 Device Failure

Device failures are handled and communicated with the metadata update routine. When a node detects a device failure it does not allow any further writes to that device until the failure has been acknowledged by all other nodes.

5. Adding a new Device

For adding a new device, it is necessary that all nodes "see" the new device to be added. For this, the following algorithm is used:

1. Node 1 issues mdadm –manage /dev/mdX –add /dev/sdYY which issues ioctl(ADD_NEW_DISK with disc.state set to MD_DISK_CLUSTER_ADD)
2. Node 1 sends a NEWDISK message with uuid and slot number
3. Other nodes issue kobject_uevent_env with uuid and slot number (Steps 4,5 could be a udev rule)
4. In userspace, the node searches for the disk, perhaps using blkid -t SUB_UUID=""
5. Other nodes issue either of the following depending on whether the disk was found: ioctl(ADD_NEW_DISK with disc.state set to MD_DISK_CANDIDATE and disc.number set to slot number) ioctl(CLUSTERED_DISK_NACK)
6. Other nodes drop lock on "no-new-devs" (CR) if device is found
7. Node 1 attempts EX lock on "no-new-dev"
8. If node 1 gets the lock, it sends METADATA_UPDATED after unmarking the disk as SpareLocal
9. If not (get "no-new-dev" lock), it fails the operation and sends METADATA_UPDATED.
10. Other nodes get the information whether a disk is added or not by the following METADATA_UPDATED.

52.1.6 6. Module interface

There are 17 call-backs which the md core can make to the cluster module. Understanding these can give a good overview of the whole process.

6.1 join(nodes) and leave()

These are called when an array is started with a clustered bitmap, and when the array is stopped. join() ensures the cluster is available and initializes the various resources. Only the first 'nodes' nodes in the cluster can use the array.

6.2 slot_number()

Reports the slot number advised by the cluster infrastructure. Range is from 0 to nodes-1.

6.3 resync_info_update()

This updates the resync range that is stored in the bitmap lock. The starting point is updated as the resync progresses. The end point is always the end of the array. It does not send a RESYNCING message.

6.4 resync_start(), resync_finish()

These are called when resync/recovery/reshape starts or stops. They update the resyncing range in the bitmap lock and also send a RESYNCING message. resync_start reports the whole array as resyncing, resync_finish reports none of it.

resync_finish() also sends a BITMAP_NEEDS_SYNC message which allows some other node to take over.

6.5 metadata_update_start(), metadata_update_finish(), metadata_update_cancel()

metadata_update_start is used to get exclusive access to the metadata. If a change is still needed once that access is gained, metadata_update_finish() will send a METADATA_UPDATE message to all other nodes, otherwise metadata_update_cancel() can be used to release the lock.

6.6 area_resyncing()

This combines two elements of functionality.

Firstly, it will check if any node is currently resyncing anything in a given range of sectors. If any resync is found, then the caller will avoid writing or read-balancing in that range.

Secondly, while node recovery is happening it reports that all areas are resyncing for READ requests. This avoids races between the cluster-filesystem and the cluster-RAID handling a node failure.

6.7 add_new_disk_start(), add_new_disk_finish(), new_disk_ack()

These are used to manage the new-disk protocol described above. When a new device is added, add_new_disk_start() is called before it is bound to the array and, if that succeeds, add_new_disk_finish() is called the device is fully added.

When a device is added in acknowledgement to a previous request, or when the device is declared "unavailable", new_disk_ack() is called.

6.8 remove_disk()

This is called when a spare or failed device is removed from the array. It causes a REMOVE message to be send to other nodes.

6.9 gather_bitmaps()

This sends a RE_ADD message to all other nodes and then gathers bitmap information from all bitmaps. This combined bitmap is then used to recovery the re-added device.

6.10 lock_all_bitmaps() and unlock_all_bitmaps()

These are called when change bitmap to none. If a node plans to clear the cluster raid's bitmap, it need to make sure no other nodes are using the raid which is achieved by lock all bitmap locks within the cluster, and also those locks are unlocked accordingly.

52.1.7 7. Unsupported features

There are somethings which are not supported by cluster MD yet.

- change array_sectors.

52.2 RAID 4/5/6 cache

Raid 4/5/6 could include an extra disk for data cache besides normal RAID disks. The role of RAID disks isn't changed with the cache disk. The cache disk caches data to the RAID disks. The cache can be in write-through (supported since 4.4) or write-back mode (supported since 4.10). mdadm (supported since 3.4) has a new option '–write-journal' to create array with cache. Please refer to mdadm manual for details. By default (RAID array starts), the cache is in write-through mode. A user can switch it to write-back mode by:

```
echo "write-back" > /sys/block/md0/md/journal_mode
```

And switch it back to write-through mode by:

```
echo "write-through" > /sys/block/md0/md/journal_mode
```

In both modes, all writes to the array will hit cache disk first. This means the cache disk must be fast and sustainable.

52.2.1 write-through mode

This mode mainly fixes the 'write hole' issue. For RAID 4/5/6 array, an unclean shutdown can cause data in some stripes to not be in consistent state, eg, data and parity don't match. The reason is that a stripe write involves several RAID disks and it's possible the writes don't hit all RAID disks yet before the unclean shutdown. We call an array degraded if it has inconsistent data. MD tries to resync the array to bring it back to normal state. But before the resync completes, any system crash will expose the chance of real data corruption in the RAID array. This problem is called 'write hole'.

The write-through cache will cache all data on cache disk first. After the data is safe on the cache disk, the data will be flushed onto RAID disks. The two-step write will guarantee MD can recover correct data after unclean shutdown even the array is degraded. Thus the cache can close the 'write hole'.

In write-through mode, MD reports IO completion to upper layer (usually filesystems) after the data is safe on RAID disks, so cache disk failure doesn't cause data loss. Of course cache disk failure means the array is exposed to 'write hole' again.

In write-through mode, the cache disk isn't required to be big. Several hundreds megabytes are enough.

52.2.2 write-back mode

write-back mode fixes the 'write hole' issue too, since all write data is cached on cache disk. But the main goal of 'write-back' cache is to speed up write. If a write crosses all RAID disks of a stripe, we call it full-stripe write. For non-full-stripe writes, MD must read old data before the new parity can be calculated. These synchronous reads hurt write throughput. Some writes which are sequential but not dispatched in the same time will suffer from this overhead too. Write-back cache will aggregate the data and flush the data to RAID disks only after the data becomes a full stripe write. This will completely avoid the overhead, so it's very helpful for some workloads. A typical workload which does sequential write followed by fsync is an example.

In write-back mode, MD reports IO completion to upper layer (usually filesystems) right after the data hits cache disk. The data is flushed to raid disks later after specific conditions met. So cache disk failure will cause data loss.

In write-back mode, MD also caches data in memory. The memory cache includes the same data stored on cache disk, so a power loss doesn't cause data loss. The memory cache size has performance impact for the array. It's recommended the size is big. A user can configure the size by:

```
echo "2048" > /sys/block/md0/md/stripe_cache_size
```

Too small cache disk will make the write aggregation less efficient in this mode depending on the workloads. It’s recommended to use a cache disk with at least several gigabytes size in write-back mode.

52.2.3 The implementation

The write-through and write-back cache use the same disk format. The cache disk is organized as a simple write log. The log consists of ‘meta data’ and ‘data’ pairs. The meta data describes the data. It also includes checksum and sequence ID for recovery identification. Data can be IO data and parity data. Data is checksumed too. The checksum is stored in the meta data ahead of the data. The checksum is an optimization because MD can write meta and data freely without worry about the order. MD superblock has a field pointed to the valid meta data of log head.

The log implementation is pretty straightforward. The difficult part is the order in which MD writes data to cache disk and RAID disks. Specifically, in write-through mode, MD calculates parity for IO data, writes both IO data and parity to the log, writes the data and parity to RAID disks after the data and parity is settled down in log and finally the IO is finished. Read just reads from raid disks as usual.

In write-back mode, MD writes IO data to the log and reports IO completion. The data is also fully cached in memory at that time, which means read must query memory cache. If some conditions are met, MD will flush the data to RAID disks. MD will calculate parity for the data and write parity into the log. After this is finished, MD will write both data and parity into RAID disks, then MD can release the memory cache. The flush conditions could be stripe becomes a full stripe write, free cache disk space is low or free in-kernel memory cache space is low.

After an unclean shutdown, MD does recovery. MD reads all meta data and data from the log. The sequence ID and checksum will help us detect corrupted meta data and data. If MD finds a stripe with data and valid parities (1 parity for raid4/5 and 2 for raid6), MD will write the data and parities to RAID disks. If parities are incompleted, they are discarded. If part of data is corrupted, they are discarded too. MD then loads valid data and writes them to RAID disks in normal way.

52.3 Partial Parity Log

Partial Parity Log (PPL) is a feature available for RAID5 arrays. The issue addressed by PPL is that after a dirty shutdown, parity of a particular stripe may become inconsistent with data on other member disks. If the array is also in degraded state, there is no way to recalculate parity, because one of the disks is missing. This can lead to silent data corruption when rebuilding the array or using it is as degraded - data calculated from parity for array blocks that have not been touched by a write request during the unclean shutdown can be incorrect. Such condition is known as the RAID5 Write Hole. Because of this, md by default does not allow starting a dirty degraded array.

Partial parity for a write operation is the XOR of stripe data chunks not modified by this write. It is just enough data needed for recovering from the write hole. XORing

partial parity with the modified chunks produces parity for the stripe, consistent with its state before the write operation, regardless of which chunk writes have completed. If one of the not modified data disks of this stripe is missing, this updated parity can be used to recover its contents. PPL recovery is also performed when starting an array after an unclean shutdown and all disks are available, eliminating the need to resync the array. Because of this, using write-intent bitmap and PPL together is not supported.

When handling a write request PPL writes partial parity before new data and parity are dispatched to disks. PPL is a distributed log - it is stored on array member drives in the metadata area, on the parity drive of a particular stripe. It does not require a dedicated journaling drive. Write performance is reduced by up to 30%-40% but it scales with the number of drives in the array and the journaling drive does not become a bottleneck or a single point of failure.

Unlike raid5-cache, the other solution in md for closing the write hole, PPL is not a true journal. It does not protect from losing in-flight data, only from silent data corruption. If a dirty disk of a stripe is lost, no PPL recovery is performed for this stripe (parity is not updated). So it is possible to have arbitrary data in the written part of a stripe if that disk is lost. In such case the behavior is the same as in plain raid5.

PPL is available for md version-1 metadata and external (specifically IMSM) metadata arrays. It can be enabled using mdadm option –consistency-policy=ppl.

There is a limitation of maximum 64 disks in the array for PPL. It allows to keep data structures and implementation simple. RAID5 arrays with so many disks are not likely due to high risk of multiple disks failure. Such restriction should not be a real life limitation.

CHAPTER

FIFTYTHREE

MEDIA SUBSYSTEM KERNEL INTERNAL API

This section contains usage information about media subsystem and its supported drivers.

Please see:

- **/admin-guide/media/index** for usage information about media subsystem and supported drivers;
- **/userspace-api/media/index** for the userspace APIs used on media devices.

53.1 Video4Linux devices

53.1.1 Introduction

The V4L2 drivers tend to be very complex due to the complexity of the hardware: most devices have multiple ICs, export multiple device nodes in /dev, and create also non-V4L2 devices such as DVB, ALSA, FB, I2C and input (IR) devices.

Especially the fact that V4L2 drivers have to setup supporting ICs to do audio/video muxing/encoding/decoding makes it more complex than most. Usually these ICs are connected to the main bridge driver through one or more I2C buses, but other buses can also be used. Such devices are called ‘sub-devices’.

For a long time the framework was limited to the video_device struct for creating V4L device nodes and video_buf for handling the video buffers (note that this document does not discuss the video_buf framework).

This meant that all drivers had to do the setup of device instances and connecting to sub-devices themselves. Some of this is quite complicated to do right and many drivers never did do it correctly.

There is also a lot of common code that could never be refactored due to the lack of a framework.

So this framework sets up the basic building blocks that all drivers need and this same framework should make it much easier to refactor common code into utility functions shared by all drivers.

A good example to look at as a reference is the v4l2-pci-skeleton.c source that is available in samples/v4l/. It is a skeleton driver for a PCI capture card, and demonstrates how to use the V4L2 driver framework. It can be used as a template for real PCI video capture driver.

53.1.2 Structure of a V4L driver

All drivers have the following structure:

1) A struct for each device instance containing the device state.
2) A way of initializing and commanding sub-devices (if any).
3) Creating V4L2 device nodes (/dev/videoX, /dev/vbiX and /dev/radioX) and keeping track of device-node specific data.
4) Filehandle-specific structs containing per-filehandle data;
5) video buffer handling.

This is a rough schematic of how it all relates:

```
device instances
  |
  +-sub-device instances
  |
  \-V4L2 device nodes
      |
      \-filehandle instances
```

53.1.3 Structure of the V4L2 framework

The framework closely resembles the driver structure: it has a v4l2_device struct for the device instance data, a v4l2_subdev struct to refer to sub-device instances, the video_device struct stores V4L2 device node data and the v4l2_fh struct keeps track of filehandle instances.

The V4L2 framework also optionally integrates with the media framework. If a driver sets the struct v4l2_device mdev field, sub-devices and video nodes will automatically appear in the media framework as entities.

53.1.4 Video device's internal representation

The actual device nodes in the `/dev` directory are created using the `video_device` struct (`v4l2-dev.h`). This struct can either be allocated dynamically or embedded in a larger struct.

To allocate it dynamically use `video_device_alloc()`:

```
struct video_device *vdev = video_device_alloc();

if (vdev == NULL)
        return -ENOMEM;

vdev->release = video_device_release;
```

If you embed it in a larger struct, then you must set the `release()` callback to your own function:

```
struct video_device *vdev = &my_vdev->vdev;

vdev->release = my_vdev_release;
```

The `release()` callback must be set and it is called when the last user of the video device exits.

The default `video_device_release()` callback currently just calls `kfree` to free the allocated memory.

There is also a `video_device_release_empty()` function that does nothing (is empty) and should be used if the struct is embedded and there is nothing to do when it is released.

You should also set these fields of `video_device`:

- `video_device->v4l2_dev`: must be set to the `v4l2_device` parent device.
- `video_device->name`: set to something descriptive and unique.
- `video_device->vfl_dir`: set this to `VFL_DIR_RX` for capture devices (`VFL_DIR_RX` has value 0, so this is normally already the default), set to `VFL_DIR_TX` for output devices and `VFL_DIR_M2M` for mem2mem (codec) devices.
- `video_device->fops`: set to the `v4l2_file_operations` struct.
- `video_device->ioctl_ops`: if you use the `v4l2_ioctl_ops` to simplify ioctl maintenance (highly recommended to use this and it might become compulsory in the future!), then set this to your `v4l2_ioctl_ops` struct. The `video_device->vfl_type` and `video_device->vfl_dir` fields are used to disable ops that do not match the type/dir combination. E.g. VBI ops are disabled for non-VBI nodes, and output ops are disabled for a capture device. This makes it possible to provide just one `v4l2_ioctl_ops` struct for both vbi and video nodes.
- `video_device->lock`: leave to `NULL` if you want to do all the locking in the driver. Otherwise you give it a pointer to a struct `mutex_lock` and before the `video_device->unlocked_ioctl` file operation is called this lock will be taken by the core and released afterwards. See the next section for more details.
- `video_device->queue`: a pointer to the struct `vb2_queue` associated with this device node. If queue is not `NULL`, and queue->lock is not `NULL`, then queue->lock is used for the queuing ioctls (`VIDIOC_REQBUFS`, `CREATE_BUFS`, `QBUF`, `DQBUF`, `QUERYBUF`, `PREPARE_BUF`, `STREAMON` and `STREAMOFF`) instead of the lock above. That way the vb2 queuing framework does not have to wait for other ioctls. This queue pointer is also used by the vb2 helper functions to check for queuing ownership (i.e. is the filehandle calling it allowed to do the operation).
- `video_device->prio`: keeps track of the priorities. Used to implement `VIDIOC_G_PRIORITY` and `VIDIOC_S_PRIORITY`. If left to `NULL`, then it will use the struct `v4l2_prio_state` in `v4l2_device`. If you want to have a separate priority state per (group of) device node(s), then you can point it to your own struct `v4l2_prio_state`.

- video_device->dev_parent: you only set this if v4l2_device was registered with NULL as the parent device struct. This only happens in cases where one hardware device has multiple PCI devices that all share the same v4l2_device core.

  The cx88 driver is an example of this: one core v4l2_device struct, but it is used by both a raw video PCI device (cx8800) and a MPEG PCI device (cx8802). Since the v4l2_device cannot be associated with two PCI devices at the same time it is setup without a parent device. But when the struct video_device is initialized you **do** know which parent PCI device to use and so you set dev_device to the correct PCI device.

If you use v4l2_ioctl_ops, then you should set video_device->unlocked_ioctl to video_ioctl2() in your v4l2_file_operations struct.

In some cases you want to tell the core that a function you had specified in your v4l2_ioctl_ops should be ignored. You can mark such ioctls by calling this function before video_register_device() is called:

    v4l2_disable_ioctl (vdev, cmd).

This tends to be needed if based on external factors (e.g. which card is being used) you want to turns off certain features in v4l2_ioctl_ops without having to make a new struct.

The v4l2_file_operations struct is a subset of file_operations. The main difference is that the inode argument is omitted since it is never used.

If integration with the media framework is needed, you must initialize the media_entity struct embedded in the video_device struct (entity field) by calling media_entity_pads_init():

```
struct media_pad *pad = &my_vdev->pad;
int err;

err = media_entity_pads_init(&vdev->entity, 1, pad);
```

The pads array must have been previously initialized. There is no need to manually set the struct media_entity type and name fields.

A reference to the entity will be automatically acquired/released when the video device is opened/closed.

ioctls and locking

The V4L core provides optional locking services. The main service is the lock field in struct video_device, which is a pointer to a mutex. If you set this pointer, then that will be used by unlocked_ioctl to serialize all ioctls.

If you are using the videobuf2 framework, then there is a second lock that you can set: video_device->queue->lock. If set, then this lock will be used instead of video_device->lock to serialize all queuing ioctls (see the previous section for the full list of those ioctls).

The advantage of using a different lock for the queuing ioctls is that for some drivers (particularly USB drivers) certain commands such as setting controls can

take a long time, so you want to use a separate lock for the buffer queuing ioctls. That way your VIDIOC_DQBUF doesn't stall because the driver is busy changing the e.g. exposure of the webcam.

Of course, you can always do all the locking yourself by leaving both lock pointers at NULL.

If you use the old videobuf framework then you must pass the video_device->lock to the videobuf queue initialize function: if videobuf has to wait for a frame to arrive, then it will temporarily unlock the lock and relock it afterwards. If your driver also waits in the code, then you should do the same to allow other processes to access the device node while the first process is waiting for something.

In the case of videobuf2 you will need to implement the wait_prepare() and wait_finish() callbacks to unlock/lock if applicable. If you use the queue->lock pointer, then you can use the helper functions vb2_ops_wait_prepare() and vb2_ops_wait_finish().

The implementation of a hotplug disconnect should also take the lock from video_device before calling v4l2_device_disconnect. If you are also using video_device->queue->lock, then you have to first lock video_device->queue->lock followed by video_device->lock. That way you can be sure no ioctl is running when you call v4l2_device_disconnect().

Video device registration

Next you register the video device with video_register_device(). This will create the character device for you.

```
err = video_register_device(vdev, VFL_TYPE_VIDEO, -1);
if (err) {
        video_device_release(vdev); /* or kfree(my_vdev); */
        return err;
}
```

If the v4l2_device parent device has a not NULL mdev field, the video device entity will be automatically registered with the media device.

Which device is registered depends on the type argument. The following types exist:

vfl_devnode_type	Device name	Usage
VFL_TYPE_VIDEO	/dev/videoX	for video input/output devices
VFL_TYPE_VBI	/dev/vbiX	for vertical blank data (i.e. closed captions, teletext)
VFL_TYPE_RADIO	/dev/radioX	for radio tuners
VFL_TYPE_SUBDEV	/dev/v4l-subdevX	for V4L2 subdevices
VFL_TYPE_SDR	/dev/swradioX	for Software Defined Radio (SDR) tuners
VFL_TYPE_TOUCH	/dev/v4l-touchX	for touch sensors

The last argument gives you a certain amount of control over the device device node number used (i.e. the X in videoX). Normally you will pass -1 to let the

v4l2 framework pick the first free number. But sometimes users want to select a specific node number. It is common that drivers allow the user to select a specific device node number through a driver module option. That number is then passed to this function and video_register_device will attempt to select that device node number. If that number was already in use, then the next free device node number will be selected and it will send a warning to the kernel log.

Another use-case is if a driver creates many devices. In that case it can be useful to place different video devices in separate ranges. For example, video capture devices start at 0, video output devices start at 16. So you can use the last argument to specify a minimum device node number and the v4l2 framework will try to pick the first free number that is equal or higher to what you passed. If that fails, then it will just pick the first free number.

Since in this case you do not care about a warning about not being able to select the specified device node number, you can call the function `video_register_device_no_warn()` instead.

Whenever a device node is created some attributes are also created for you. If you look in `/sys/class/video4linux` you see the devices. Go into e.g. `video0` and you will see 'name', 'dev_debug' and 'index' attributes. The 'name' attribute is the 'name' field of the video_device struct. The 'dev_debug' attribute can be used to enable core debugging. See the next section for more detailed information on this.

The 'index' attribute is the index of the device node: for each call to `video_register_device()` the index is just increased by 1. The first video device node you register always starts with index 0.

Users can setup udev rules that utilize the index attribute to make fancy device names (e.g. '`mpegX`' for MPEG video capture device nodes).

After the device was successfully registered, then you can use these fields:

- `video_device->`vfl_type: the device type passed to `video_register_device()`.
- `video_device->`minor: the assigned device minor number.
- `video_device->`num: the device node number (i.e. the X in `videoX`).
- `video_device->`index: the device index number.

If the registration failed, then you need to call `video_device_release()` to free the allocated `video_device` struct, or free your own struct if the `video_device` was embedded in it. The `vdev->release()` callback will never be called if the registration failed, nor should you ever attempt to unregister the device if the registration failed.

video device debugging

The 'dev_debug' attribute that is created for each video, vbi, radio or swradio device in `/sys/class/video4linux/<devX>/` allows you to enable logging of file operations.

It is a bitmask and the following bits can be set:

Mask	Description
0x01	Log the ioctl name and error code. VIDIOC_(D)QBUF ioctls are only logged if bit 0x08 is also set.
0x02	Log the ioctl name arguments and error code. VIDIOC_(D)QBUF ioctls are only logged if bit 0x08 is also set.
0x04	Log the file operations open, release, read, write, mmap and get_unmapped_area. The read and write operations are only logged if bit 0x08 is also set.
0x08	Log the read and write file operations and the VIDIOC_QBUF and VIDIOC_DQBUF ioctls.
0x10	Log the poll file operation.
0x20	Log error and messages in the control operations.

Video device cleanup

When the video device nodes have to be removed, either during the unload of the driver or because the USB device was disconnected, then you should unregister them with:

```
video_unregister_device() (vdev);
```

This will remove the device nodes from sysfs (causing udev to remove them from `/dev`).

After `video_unregister_device()` returns no new opens can be done. However, in the case of USB devices some application might still have one of these device nodes open. So after the unregister all file operations (except release, of course) will return an error as well.

When the last user of the video device node exits, then the `vdev->release()` callback is called and you can do the final cleanup there.

Don't forget to cleanup the media entity associated with the video device if it has been initialized:

```
media_entity_cleanup (&vdev->entity);
```

This can be done from the release callback.

helper functions

There are a few useful helper functions:

- file and `video_device` private data

You can set/get driver private data in the video_device struct using:

`video_get_drvdata` (`vdev`);

`video_set_drvdata` (`vdev`);

Note that you can safely call `video_set_drvdata()` before calling `video_register_device()`.

And this function:

`video_devdata` (struct file *file);

returns the video_device belonging to the file struct.

The `video_devdata()` function combines `video_get_drvdata()` with `video_devdata()`:

`video_drvdata` (struct file *file);

You can go from a `video_device` struct to the v4l2_device struct using:

```
struct v4l2_device *v4l2_dev = vdev->v4l2_dev;
```

- Device node name

The `video_device` node kernel name can be retrieved using:

`video_device_node_name` (`vdev`);

The name is used as a hint by userspace tools such as udev. The function should be used where possible instead of accessing the video_device::num and video_device::minor fields.

video_device functions and data structures

enum **vfl_devnode_type**
type of V4L2 device node

Constants

VFL_TYPE_VIDEO for video input/output devices

VFL_TYPE_VBI for vertical blank data (i.e. closed captions, teletext)

VFL_TYPE_RADIO for radio tuners

VFL_TYPE_SUBDEV for V4L2 subdevices

VFL_TYPE_SDR for Software Defined Radio tuners

VFL_TYPE_TOUCH for touch sensors

VFL_TYPE_MAX number of VFL types, must always be last in the enum

enum **vfl_devnode_direction**
Identifies if a `struct video_device` corresponds to a receiver, a transmitter or a mem-to-mem device.

Constants

VFL_DIR_RX device is a receiver.

VFL_DIR_TX device is a transmitter.

VFL_DIR_M2M device is a memory to memory device.

Note

Ignored if `enum vfl_devnode_type` is `VFL_TYPE_SUBDEV`.

enum **v4l2_video_device_flags**
Flags used by `struct video_device`

Constants

V4L2_FL_REGISTERED indicates that a `struct video_device` is registered. Drivers can clear this flag if they want to block all future device access. It is cleared by video_unregister_device.

V4L2_FL_USES_V4L2_FH indicates that file->private_data points to `struct v4l2_fh`. This flag is set by the core when `v4l2_fh_init()` is called. All new drivers should use it.

V4L2_FL_QUIRK_INVERTED_CROP some old M2M drivers use g/s_crop/cropcap incorrectly: crop and compose are swapped. If this flag is set, then the selection targets are swapped in the g/s_crop/cropcap functions in v4l2-ioctl.c. This allows those drivers to correctly implement the selection API, but the old crop API will still work as expected in order to preserve backwards compatibility. Never set this flag for new drivers.

V4L2_FL_SUBDEV_RO_DEVNODE indicates that the video device node is registered in read-only mode. The flag only applies to device nodes registered for subdevices, it is set by the core when the sub-devices device nodes are registered with `v4l2_device_register_ro_subdev_nodes()` and used by the subdevice ioctl handler to restrict access to some ioctl calls.

struct **v4l2_prio_state**
stores the priority states

Definition

```
struct v4l2_prio_state {
  atomic_t prios[4];
};
```

Members

prios array with elements to store the array priorities

Description

Note: The size of **prios** array matches the number of priority types defined by

enum v4l2_priority.

void **v4l2_prio_init**(struct v4l2_prio_state * global)
initializes a struct v4l2_prio_state

Parameters

struct v4l2_prio_state * global pointer to struct v4l2_prio_state

int **v4l2_prio_change**(struct v4l2_prio_state * global, enum v4l2_priority * local, enum v4l2_priority new)
changes the v4l2 file handler priority

Parameters

struct v4l2_prio_state * global pointer to the struct v4l2_prio_state of the device node.

enum v4l2_priority * local pointer to the desired priority, as defined by enum v4l2_priority

enum v4l2_priority new Priority type requested, as defined by enum v4l2_priority.

Description

Note: This function should be used only by the V4L2 core.

void **v4l2_prio_open**(struct v4l2_prio_state * global, enum v4l2_priority * local)
Implements the priority logic for a file handler open

Parameters

struct v4l2_prio_state * global pointer to the struct v4l2_prio_state of the device node.

enum v4l2_priority * local pointer to the desired priority, as defined by enum v4l2_priority

Description

Note: This function should be used only by the V4L2 core.

void **v4l2_prio_close**(struct v4l2_prio_state * global, enum v4l2_priority local)
Implements the priority logic for a file handler close

Parameters

struct v4l2_prio_state * global pointer to the struct v4l2_prio_state of the device node.

enum v4l2_priority local priority to be released, as defined by enum v4l2_priority

Description

Note: This function should be used only by the V4L2 core.

enum v4l2_priority **v4l2_prio_max**(struct v4l2_prio_state * global)
Return the maximum priority, as stored at the **global** array.

Parameters

struct v4l2_prio_state * global pointer to the struct v4l2_prio_state of the device node.

Description

Note: This function should be used only by the V4L2 core.

int **v4l2_prio_check**(struct v4l2_prio_state * global, enum v4l2_priority local)
Implements the priority logic for a file handler close

Parameters

struct v4l2_prio_state * global pointer to the struct v4l2_prio_state of the device node.

enum v4l2_priority local desired priority, as defined by enum v4l2_priority local

Description

Note: This function should be used only by the V4L2 core.

struct **v4l2_file_operations**
fs operations used by a V4L2 device

Definition

```
struct v4l2_file_operations {
  struct module *owner;
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  __poll_t (*poll) (struct file *, struct poll_table_struct *);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
#ifdef CONFIG_COMPAT;
  long (*compat_ioctl32) (struct file *, unsigned int, unsigned long);
#endif;
  unsigned long (*get_unmapped_area) (struct file *, unsigned long,␣
 ↪unsigned long, unsigned long, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct file *);
  int (*release) (struct file *);
};
```

Members

owner pointer to struct module

read operations needed to implement the read() syscall

write operations needed to implement the write() syscall

poll operations needed to implement the poll() syscall

unlocked_ioctl operations needed to implement the ioctl() syscall

compat_ioctl32 operations needed to implement the ioctl() syscall for the special case where the Kernel uses 64 bits instructions, but the userspace uses 32 bits.

get_unmapped_area called by the mmap() syscall, used when %!CONFIG_MMU

mmap operations needed to implement the mmap() syscall

open operations needed to implement the open() syscall

release operations needed to implement the release() syscall

Description

Note: Those operations are used to implemente the fs struct file_operations at the V4L2 drivers. The V4L2 core overrides the fs ops with some extra logic needed by the subsystem.

---

struct **video_device**
Structure used to create and manage the V4L2 device nodes.

Definition

```
struct video_device {
#if defined(CONFIG_MEDIA_CONTROLLER);
  struct media_entity entity;
  struct media_intf_devnode *intf_devnode;
  struct media_pipeline pipe;
#endif;
  const struct v4l2_file_operations *fops;
  u32 device_caps;
  struct device dev;
  struct cdev *cdev;
  struct v4l2_device *v4l2_dev;
  struct device *dev_parent;
  struct v4l2_ctrl_handler *ctrl_handler;
  struct vb2_queue *queue;
  struct v4l2_prio_state *prio;
  char name[32];
  enum vfl_devnode_type vfl_type;
  enum vfl_devnode_direction vfl_dir;
  int minor;
  u16 num;
  unsigned long flags;
  int index;
  spinlock_t fh_lock;
  struct list_head        fh_list;
  int dev_debug;
```

(continues on next page)

(continued from previous page)

```
  v4l2_std_id tvnorms;
  void (*release)(struct video_device *vdev);
  const struct v4l2_ioctl_ops *ioctl_ops;
  unsigned long valid_ioctls[BITS_TO_LONGS(BASE_VIDIOC_PRIVATE)];
  struct mutex *lock;
};
```

Members

entity struct media_entity

intf_devnode pointer to struct media_intf_devnode

pipe struct media_pipeline

fops pointer to struct v4l2_file_operations for the video device

device_caps device capabilities as used in v4l2_capabilities

dev struct device for the video device

cdev character device

v4l2_dev pointer to struct v4l2_device parent

dev_parent pointer to struct device parent

ctrl_handler Control handler associated with this device node. May be NULL.

queue struct vb2_queue associated with this device node. May be NULL.

prio pointer to struct v4l2_prio_state with device's Priority state. If NULL, then v4l2_dev->prio will be used.

name video device name

vfl_type V4L device type, as defined by enum vfl_devnode_type

vfl_dir V4L receiver, transmitter or m2m

minor device node 'minor'. It is set to -1 if the registration failed

num number of the video device node

flags video device flags. Use bitops to set/clear/test flags. Contains a set of enum v4l2_video_device_flags.

index attribute to differentiate multiple indices on one physical device

fh_lock Lock for all v4l2_fhs

fh_list List of struct v4l2_fh

dev_debug Internal device debug flags, not for use by drivers

tvnorms Supported tv norms

release video device release() callback

ioctl_ops pointer to struct v4l2_ioctl_ops with ioctl callbacks

valid_ioctls bitmap with the valid ioctls for this device

lock pointer to struct mutex serialization lock

Description

Note: Only set **dev_parent** if that can't be deduced from **v4l2_dev**.

---

media_entity_to_video_device(__entity)
Returns a `struct video_device` from the `struct media_entity` embedded on it.

Parameters

__entity pointer to `struct media_entity`

to_video_device(cd)
Returns a `struct video_device` from the `struct device` embedded on it.

Parameters

cd pointer to `struct device`

int **__video_register_device**(struct video_device * vdev, enum vfl_devnode_type type, int nr, int warn_if_nr_in_use, struct module * owner)
register video4linux devices

Parameters

struct video_device * vdev struct video_device to register

enum vfl_devnode_type type type of device to register, as defined by `enum vfl_devnode_type`

int nr which device node number is desired: (0 == /dev/video0, 1 == /dev/video1, ⋯, -1 == first free)

int warn_if_nr_in_use warn if the desired device node number was already in use and another number was chosen instead.

struct module * owner module that owns the video device node

Description

The registration code assigns minor numbers and device node numbers based on the requested type and registers the new device node with the kernel.

This function assumes that struct video_device was zeroed when it was allocated and does not contain any stale date.

An error is returned if no free minor or device node number could be found, or if the registration of the device node failed.

Returns 0 on success.

Note: This function is meant to be used only inside the V4L2 core. Drivers should use `video_register_device()` or `video_register_device_no_warn()`.

---

int **video_register_device**(struct video_device * vdev, enum vfl_devnode_type type, int nr)
register video4linux devices

Parameters

struct video_device * vdev struct video_device to register

enum vfl_devnode_type type type of device to register, as defined by enum vfl_devnode_type

int nr which device node number is desired: (0 == /dev/video0, 1 == /dev/video1, ⋯, -1 == first free)

Description

Internally, it calls `__video_register_device()`. Please see its documentation for more details.

Note: if video_register_device fails, the release() callback of `struct video_device` structure is not called, so the caller is responsible for freeing any data. Usually that means that you `video_device_release()` should be called on failure.

---

int **video_register_device_no_warn**(struct video_device * vdev, enum vfl_devnode_type type, int nr)
register video4linux devices

Parameters

struct video_device * vdev struct video_device to register

enum vfl_devnode_type type type of device to register, as defined by enum vfl_devnode_type

int nr which device node number is desired: (0 == /dev/video0, 1 == /dev/video1, ⋯, -1 == first free)

Description

This function is identical to `video_register_device()` except that no warning is issued if the desired device node number was already in use.

Internally, it calls `__video_register_device()`. Please see its documentation for more details.

Note: if video_register_device fails, the release() callback of `struct video_device` structure is not called, so the caller is responsible for freeing any data. Usually that means that you `video_device_release()` should be called on failure.

---

void **video_unregister_device**(struct video_device * vdev)
Unregister video devices.

Parameters

struct video_device * vdev `struct video_device` to register

Description

Does nothing if vdev == NULL or if `video_is_registered()` returns false.

struct video_device * **video_device_alloc**(void)
helper function to alloc `struct video_device`

Parameters

void no arguments

Description

Returns NULL if `-ENOMEM` or a `struct video_device` on success.

void **video_device_release**(struct video_device * vdev)
helper function to release `struct video_device`

Parameters

struct video_device * vdev pointer to `struct video_device`

Description

Can also be used for video_device->release().

void **video_device_release_empty**(struct video_device * vdev)
helper function to implement the video_device->release() callback.

Parameters

struct video_device * vdev pointer to `struct video_device`

Description

This release function does nothing.

It should be used when the video_device is a static global struct.

Note: Having a static video_device is a dubious construction at best.

---

void **v4l2_disable_ioctl**(struct video_device * vdev, unsigned int cmd)
mark that a given command isn't implemented. shouldn't use core locking

Parameters

struct video_device * vdev pointer to `struct video_device`

unsigned int cmd ioctl command

Description

This function allows drivers to provide just one v4l2_ioctl_ops struct, but disable ioctls based on the specific card that is actually found.

Note: This must be called before video_register_device. See also the comments for determine_valid_ioctls().

---

void * **video_get_drvdata**(struct video_device * vdev)
gets private data from `struct video_device`.

Parameters

struct video_device * vdev pointer to struct video_device

Description

returns a pointer to the private data

void **video_set_drvdata**(struct video_device * vdev, void * data)
sets private data from struct video_device.

Parameters

struct video_device * vdev pointer to struct video_device

void * data private data pointer

struct video_device * **video_devdata**(struct file * file)
gets struct video_device from struct file.

Parameters

struct file * file pointer to struct file

void * **video_drvdata**(struct file * file)
gets private data from struct video_device using the struct file.

Parameters

struct file * file pointer to struct file

Description

This is function combines both video_get_drvdata() and video_devdata() as this is used very often.

const char * **video_device_node_name**(struct video_device * vdev)
returns the video device name

Parameters

struct video_device * vdev pointer to struct video_device

Description

Returns the device name string

int **video_is_registered**(struct video_device * vdev)
returns true if the struct video_device is registered.

Parameters

struct video_device * vdev pointer to struct video_device

Description

53.1.5 V4L2 device instance

Each device instance is represented by a struct `v4l2_device`. Very simple devices can just allocate this struct, but most of the time you would embed this struct inside a larger struct.

You must register the device instance by calling:

`v4l2_device_register` (dev, `v4l2_dev`).

Registration will initialize the `v4l2_device` struct. If the dev->driver_data field is `NULL`, it will be linked to `v4l2_dev` argument.

Drivers that want integration with the media device framework need to set dev->driver_data manually to point to the driver-specific device structure that embed the struct `v4l2_device` instance. This is achieved by a `dev_set_drvdata()` call before registering the V4L2 device instance. They must also set the struct `v4l2_device` mdev field to point to a properly initialized and registered `media_device` instance.

If `v4l2_dev->`name is empty then it will be set to a value derived from dev (driver name followed by the bus_id, to be precise). If you set it up before calling `v4l2_device_register()` then it will be untouched. If dev is `NULL`, then you **must** setup `v4l2_dev->`name before calling `v4l2_device_register()`.

You can use `v4l2_device_set_name()` to set the name based on a driver name and a driver-global atomic_t instance. This will generate names like `ivtv0`, `ivtv1`, etc. If the name ends with a digit, then it will insert a dash: `cx18-0`, `cx18-1`, etc. This function returns the instance number.

The first `dev` argument is normally the `struct device` pointer of a `pci_dev`, `usb_interface` or `platform_device`. It is rare for dev to be `NULL`, but it happens with ISA devices or when one device creates multiple PCI devices, thus making it impossible to associate `v4l2_dev` with a particular parent.

You can also supply a `notify()` callback that can be called by sub-devices to notify you of events. Whether you need to set this depends on the sub-device. Any notifications a sub-device supports must be defined in a header in `include/media/subdevice.h`.

V4L2 devices are unregistered by calling:

`v4l2_device_unregister()` (`v4l2_dev`).

If the dev->driver_data field points to `v4l2_dev`, it will be reset to `NULL`. Unregistering will also automatically unregister all subdevs from the device.

If you have a hotpluggable device (e.g. a USB device), then when a disconnect happens the parent device becomes invalid. Since `v4l2_device` has a pointer to that parent device it has to be cleared as well to mark that the parent is gone. To do this call:

`v4l2_device_disconnect()` (`v4l2_dev`).

This does not unregister the subdevs, so you still need to call the `v4l2_device_unregister()` function for that. If your driver is not hotpluggable, then there is no need to call `v4l2_device_disconnect()`.

Sometimes you need to iterate over all devices registered by a specific driver. This is usually the case if multiple device drivers use the same hardware. E.g. the ivtvfb driver is a framebuffer driver that uses the ivtv hardware. The same is true for alsa drivers for example.

You can iterate over all registered devices as follows:

```
static int callback(struct device *dev, void *p)
{
        struct v4l2_device *v4l2_dev = dev_get_drvdata(dev);

        /* test if this device was inited */
        if (v4l2_dev == NULL)
                return 0;
        ...
        return 0;
}

int iterate(void *p)
{
        struct device_driver *drv;
        int err;

        /* Find driver 'ivtv' on the PCI bus.
        pci_bus_type is a global. For USB buses use usb_bus_type. */
        drv = driver_find("ivtv", &pci_bus_type);
        /* iterate over all ivtv device instances */
        err = driver_for_each_device(drv, NULL, p, callback);
        put_driver(drv);
        return err;
}
```

Sometimes you need to keep a running counter of the device instance. This is commonly used to map a device instance to an index of a module option array.

The recommended approach is as follows:

```
static atomic_t drv_instance = ATOMIC_INIT(0);

static int drv_probe(struct pci_dev *pdev, const struct pci_device_id *pci_
↪id)
{
        ...
        state->instance = atomic_inc_return(&drv_instance) - 1;
}
```

If you have multiple device nodes then it can be difficult to know when it is safe to unregister `v4l2_device` for hotpluggable devices. For this purpose `v4l2_device` has refcounting support. The refcount is increased whenever `video_register_device()` is called and it is decreased whenever that device node is released. When the refcount reaches zero, then the `v4l2_device` release() callback is called. You can do your final cleanup there.

If other device nodes (e.g. ALSA) are created, then you can increase and decrease the refcount manually as well by calling:

`v4l2_device_get()` (`v4l2_dev`).

or:

v4l2_device_put() (v4l2_dev).

Since the initial refcount is 1 you also need to call v4l2_device_put() in the disconnect() callback (for USB devices) or in the remove() callback (for e.g. PCI devices), otherwise the refcount will never reach 0.

v4l2_device functions and data structures

struct **v4l2_device**
main struct to for V4L2 device drivers

Definition

```
struct v4l2_device {
  struct device *dev;
  struct media_device *mdev;
  struct list_head subdevs;
  spinlock_t lock;
  char name[V4L2_DEVICE_NAME_SIZE];
  void (*notify)(struct v4l2_subdev *sd, unsigned int notification, void␣
↪*arg);
  struct v4l2_ctrl_handler *ctrl_handler;
  struct v4l2_prio_state prio;
  struct kref ref;
  void (*release)(struct v4l2_device *v4l2_dev);
};
```

Members

dev pointer to struct device.

mdev pointer to struct media_device, may be NULL.

subdevs used to keep track of the registered subdevs

lock lock this struct; can be used by the driver as well if this struct is embedded into a larger struct.

name unique device name, by default the driver name + bus ID

notify notify operation called by some sub-devices.

ctrl_handler The control handler. May be NULL.

prio Device' s priority state

ref Keep track of the references to this struct.

release Release function that is called when the ref count goes to 0.

Description

Each instance of a V4L2 device should create the v4l2_device struct, either stand-alone or embedded in a larger struct.

It allows easy access to sub-devices (see v4l2-subdev.h) and provides basic V4L2 device-level support.

Note:

1) **dev->driver_data** points to this struct.

2) **dev** might be NULL if there is no parent device

void **v4l2_device_get**(struct v4l2_device * v4l2_dev)
gets a V4L2 device reference

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

Description

This is an ancillary routine meant to increment the usage for the struct v4l2_device pointed by **v4l2_dev**.

int **v4l2_device_put**(struct v4l2_device * v4l2_dev)
puts a V4L2 device reference

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

Description

This is an ancillary routine meant to decrement the usage for the struct v4l2_device pointed by **v4l2_dev**.

int **v4l2_device_register**(struct device * dev, struct v4l2_device * v4l2_dev)
Initialize v4l2_dev and make **dev->driver_data** point to **v4l2_dev**.

Parameters

struct device * dev pointer to struct device

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

Description

Note: dev may be NULL in rare cases (ISA devices). In such case the caller must fill in the **v4l2_dev->name** field before calling this function.

int **v4l2_device_set_name**(struct v4l2_device * v4l2_dev, const char * basename, atomic_t * instance)
Optional function to initialize the name field of struct v4l2_device

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

const char * basename base name for the device name

atomic_t * instance pointer to a static atomic_t var with the instance usage for the device driver.

Description

`v4l2_device_set_name()` initializes the name field of struct `v4l2_device` using the driver name and a driver-global atomic_t instance.

This function will increment the instance counter and returns the instance value used in the name.

The first time this is called the name field will be set to foo0 and this function returns 0. If the name ends with a digit (e.g. cx18), then the name will be set to cx18-0 since cx180 would look really odd.

Example

```
static atomic_t drv_instance = ATOMIC_INIT(0);
...
instance = v4l2_device_set_name(&v4l2_dev, "foo", &drv_instance);
```

void **v4l2_device_disconnect**(struct v4l2_device * v4l2_dev)
Change V4L2 device state to disconnected.

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

Description

Should be called when the USB parent disconnects. Since the parent disappears, this ensures that **v4l2_dev** doesn't have an invalid parent pointer.

Note: This function sets **v4l2_dev->dev** to NULL.

---

void **v4l2_device_unregister**(struct v4l2_device * v4l2_dev)
Unregister all sub-devices and any other resources related to **v4l2_dev**.

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

int **v4l2_device_register_subdev**(struct v4l2_device * v4l2_dev, struct v4l2_subdev * sd)
Registers a subdev with a v4l2 device.

Parameters

struct v4l2_device * v4l2_dev pointer to struct `v4l2_device`

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

Description

While registered, the subdev module is marked as in-use.

An error is returned if the module is no longer loaded on any attempts to register it.

void **v4l2_device_unregister_subdev**(struct v4l2_subdev * sd)
Unregisters a subdev with a v4l2 device.

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

Description

Note: Can also be called if the subdev wasn't registered. In such case, it will do nothing.

---

int **__v4l2_device_register_subdev_nodes**(struct v4l2_device * v4l2_dev, bool read_only)

Registers device nodes for all subdevs of the v4l2 device that are marked with the `V4L2_SUBDEV_FL_HAS_DEVNODE` flag.

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

bool read_only subdevices read-only flag. True to register the subdevices device nodes in read-only mode, false to allow full access to the subdevice userspace API.

int **v4l2_device_register_subdev_nodes**(struct v4l2_device * v4l2_dev)

Registers subdevices device nodes with unrestricted access to the subdevice userspace operations

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

Description

Internally calls `__v4l2_device_register_subdev_nodes()`. See its documentation for more details.

int **v4l2_device_register_ro_subdev_nodes**(struct v4l2_device * v4l2_dev)

Registers subdevices device nodes in read-only mode

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

Description

Internally calls `__v4l2_device_register_subdev_nodes()`. See its documentation for more details.

void **v4l2_subdev_notify**(struct v4l2_subdev * sd, unsigned int notification, void * arg)

Sends a notification to v4l2_device.

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

unsigned int notification type of notification. Please notice that the notification type is driver-specific.

void * arg arguments for the notification. Those are specific to each notification type.

bool **v4l2_device_supports_requests**(struct v4l2_device * v4l2_dev)
Test if requests are supported.

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

v4l2_device_for_each_subdev(sd, v4l2_dev)
Helper macro that interates over all sub-devices of a given `v4l2_device`.

Parameters

sd pointer that will be filled by the macro with all `struct v4l2_subdev` pointer used as an iterator by the loop.

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

Description

This macro iterates over all sub-devices owned by the **v4l2_dev** device. It acts as a for loop iterator and executes the next statement with the **sd** variable pointing to each sub-device in turn.

__v4l2_device_call_subdevs_p(v4l2_dev, sd, cond, o, f, args)
Calls the specified operation for all subdevs matching the condition.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

sd pointer that will be filled by the macro with all `struct v4l2_subdev` pointer used as an iterator by the loop.

cond condition to be match

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Description

Ignore any errors.

Note

subdevs cannot be added or deleted while walking the subdevs list.

__v4l2_device_call_subdevs(v4l2_dev, cond, o, f, args)
Calls the specified operation for all subdevs matching the condition.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

cond condition to be match

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Description

Ignore any errors.

Note

subdevs cannot be added or deleted while walking the subdevs list.

__v4l2_device_call_subdevs_until_err_p(v4l2_dev, sd, cond, o, f, args)
Calls the specified operation for all subdevs matching the condition.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

sd pointer that will be filled by the macro with all `struct v4l2_subdev` sub-devices associated with **v4l2_dev**.

cond condition to be match

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Return

Description

If the operation returns an error other than 0 or `-ENOIOCTLCMD` for any subdevice, then abort and return with that error code, zero otherwise.

Note

subdevs cannot be added or deleted while walking the subdevs list.

__v4l2_device_call_subdevs_until_err(v4l2_dev, cond, o, f, args)
Calls the specified operation for all subdevs matching the condition.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

cond condition to be match

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Return

Description

If the operation returns an error other than 0 or `-ENOIOCTLCMD` for any subdevice, then abort and return with that error code, zero otherwise.

Note

subdevs cannot be added or deleted while walking the subdevs list.

v4l2_device_call_all(v4l2_dev, grpid, o, f, args)
Calls the specified operation for all subdevs matching the `v4l2_subdev.grp_id`, as assigned by the bridge driver.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

grpid `struct v4l2_subdev->grp_id` group ID to match. Use 0 to match them all.

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Description

Ignore any errors.

Note

subdevs cannot be added or deleted while walking the subdevs list.

v4l2_device_call_until_err(v4l2_dev, grpid, o, f, args)
Calls the specified operation for all subdevs matching the `v4l2_subdev.grp_id`, as assigned by the bridge driver, until an error occurs.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

grpid `struct v4l2_subdev->grp_id` group ID to match. Use 0 to match them all.

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Return

Description

If the operation returns an error other than 0 or `-ENOIOCTLCMD` for any subdevice, then abort and return with that error code, zero otherwise.

Note

subdevs cannot be added or deleted while walking the subdevs list.

v4l2_device_mask_call_all(v4l2_dev, grpmsk, o, f, args)
Calls the specified operation for all subdevices where a group ID matches a specified bitmask.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

grpmsk bitmask to be checked against `struct v4l2_subdev`->grp_id group ID to be matched. Use 0 to match them all.

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Description

Ignore any errors.

Note

subdevs cannot be added or deleted while walking the subdevs list.

v4l2_device_mask_call_until_err(v4l2_dev, grpmsk, o, f, args)
Calls the specified operation for all subdevices where a group ID matches a specified bitmask.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

grpmsk bitmask to be checked against `struct v4l2_subdev`->grp_id group ID to be matched. Use 0 to match them all.

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Return

Description

If the operation returns an error other than 0 or `-ENOIOCTLCMD` for any subdevice, then abort and return with that error code, zero otherwise.

Note

subdevs cannot be added or deleted while walking the subdevs list.

v4l2_device_has_op(v4l2_dev, grpid, o, f)
checks if any subdev with matching grpid has a given ops.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

grpid `struct v4l2_subdev->grp_id` group ID to match. Use 0 to match them all.

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

v4l2_device_mask_has_op(v4l2_dev, grpmsk, o, f)
checks if any subdev with matching group mask has a given ops.

Parameters

v4l2_dev `struct v4l2_device` owning the sub-devices to iterate over.

grpmsk bitmask to be checked against `struct v4l2_subdev->grp_id` group ID to be matched. Use 0 to match them all.

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of operations functions.

f operation function that will be called if **cond** matches. The operation functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

53.1.6 V4L2 File handlers

struct `v4l2_fh` provides a way to easily keep file handle specific data that is used by the V4L2 framework.

> **Attention:** New drivers must use struct `v4l2_fh` since it is also used to implement priority handling (VIDIOC_G_PRIORITY).

The users of `v4l2_fh` (in the V4L2 framework, not the driver) know whether a driver uses `v4l2_fh` as its `file->private_data` pointer by testing the `V4L2_FL_USES_V4L2_FH` bit in `video_device->flags`. This bit is set whenever `v4l2_fh_init()` is called.

struct `v4l2_fh` is allocated as a part of the driver's own file handle structure and `file->private_data` is set to it in the driver's `open()` function by the driver.

In many cases the struct `v4l2_fh` will be embedded in a larger structure. In that case you should call:

1) `v4l2_fh_init()` and `v4l2_fh_add()` in `open()`
2) `v4l2_fh_del()` and `v4l2_fh_exit()` in `release()`

Drivers can extract their own file handle structure by using the container_of macro.

Example:

```
struct my_fh {
        int blah;
        struct v4l2_fh fh;
};

...

int my_open(struct file *file)
{
        struct my_fh *my_fh;
        struct video_device *vfd;
        int ret;

        ...

        my_fh = kzalloc(sizeof(*my_fh), GFP_KERNEL);

        ...

        v4l2_fh_init(&my_fh->fh, vfd);

        ...

        file->private_data = &my_fh->fh;
        v4l2_fh_add(&my_fh->fh);
        return 0;
}

int my_release(struct file *file)
{
        struct v4l2_fh *fh = file->private_data;
        struct my_fh *my_fh = container_of(fh, struct my_fh, fh);

        ...
        v4l2_fh_del(&my_fh->fh);
        v4l2_fh_exit(&my_fh->fh);
        kfree(my_fh);
        return 0;
}
```

Below is a short description of the v4l2_fh functions used:

v4l2_fh_init (fh, vdev)

- Initialise the file handle. This **MUST** be performed in the driver's v4l2_file_operations->open() handler.

v4l2_fh_add (fh)

- Add a v4l2_fh to video_device file handle list. Must be called once the file handle is completely initialized.

v4l2_fh_del (fh)

- Unassociate the file handle from video_device. The file handle exit function may now be called.

v4l2_fh_exit (fh)

- Uninitialise the file handle. After uninitialisation the v4l2_fh memory can be freed.

If struct v4l2_fh is not embedded, then you can use these helper functions:

v4l2_fh_open (struct file *filp)

- This allocates a struct v4l2_fh, initializes it and adds it to the struct video_device associated with the file struct.

v4l2_fh_release (struct file *filp)

- This deletes it from the struct video_device associated with the file struct, uninitialised the v4l2_fh and frees it.

These two functions can be plugged into the v4l2_file_operation's open() and release() ops.

Several drivers need to do something when the first file handle is opened and when the last file handle closes. Two helper functions were added to check whether the v4l2_fh struct is the only open filehandle of the associated device node:

v4l2_fh_is_singular (fh)

- Returns 1 if the file handle is the only open file handle, else 0.

v4l2_fh_is_singular_file (struct file *filp)

- Same, but it calls v4l2_fh_is_singular with filp->private_data.

V4L2 fh functions and data structures

struct **v4l2_fh**
Describes a V4L2 file handler

Definition

```
struct v4l2_fh {
  struct list_head        list;
  struct video_device     *vdev;
  struct v4l2_ctrl_handler *ctrl_handler;
  enum v4l2_priority      prio;
  wait_queue_head_t wait;
  struct mutex            subscribe_lock;
  struct list_head        subscribed;
  struct list_head        available;
  unsigned int            navailable;
  u32 sequence;
  struct v4l2_m2m_ctx     *m2m_ctx;
};
```

Members

list list of file handlers

vdev pointer to struct video_device

ctrl_handler pointer to struct v4l2_ctrl_handler

prio priority of the file handler, as defined by `enum v4l2_priority`

wait event's wait queue

subscribe_lock serialise changes to the subscribed list; guarantee that the add and del event callbacks are orderly called

subscribed list of subscribed events

available list of events waiting to be dequeued

navailable number of available events at **available** list

sequence event sequence number

m2m_ctx pointer to `struct v4l2_m2m_ctx`

void **v4l2_fh_init**(struct v4l2_fh * fh, struct video_device * vdev)
Initialise the file handle.

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

struct video_device * vdev pointer to `struct video_device`

Description

Parts of the V4L2 framework using the file handles should be initialised in this function. Must be called from driver's v4l2_file_operations->open() handler if the driver uses `struct v4l2_fh`.

void **v4l2_fh_add**(struct v4l2_fh * fh)
Add the fh to the list of file handles on a video_device.

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

Description

Note: The **fh** file handle must be initialised first.

---

int **v4l2_fh_open**(struct file * filp)
Ancillary routine that can be used as the open() op of v4l2_file_operations.

Parameters

struct file * filp pointer to struct file

Description

It allocates a v4l2_fh and inits and adds it to the `struct video_device` associated with the file pointer.

void **v4l2_fh_del**(struct v4l2_fh * fh)
Remove file handle from the list of file handles.

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

Description

On error filp->private_data will be `NULL`, otherwise it will point to the `struct v4l2_fh`.

Note: Must be called in v4l2_file_operations->release() handler if the driver uses `struct v4l2_fh`.

---

void **v4l2_fh_exit**(struct v4l2_fh * fh)
Release resources related to a file handle.

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

Description

Parts of the V4L2 framework using the v4l2_fh must release their resources here, too.

Note: Must be called in v4l2_file_operations->release() handler if the driver uses `struct v4l2_fh`.

---

int **v4l2_fh_release**(struct file * filp)
Ancillary routine that can be used as the release() op of v4l2_file_operations.

Parameters

struct file * filp pointer to struct file

Description

It deletes and exits the v4l2_fh associated with the file pointer and frees it. It will do nothing if filp->private_data (the pointer to the v4l2_fh struct) is `NULL`.

This function always returns 0.

int **v4l2_fh_is_singular**(struct v4l2_fh * fh)
Returns 1 if this filehandle is the only filehandle opened for the associated video_device.

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

Description

If **fh** is NULL, then it returns 0.

int **v4l2_fh_is_singular_file**(struct file * filp)
Returns 1 if this filehandle is the only filehandle opened for the associated video_device.

Parameters

struct file * filp pointer to struct file

Description

This is a helper function variant of v4l2_fh_is_singular() with uses struct file as argument.

If filp->private_data is NULL, then it will return 0.

53.1.7 V4L2 sub-devices

Many drivers need to communicate with sub-devices. These devices can do all sort of tasks, but most commonly they handle audio and/or video muxing, encoding or decoding. For webcams common sub-devices are sensors and camera controllers.

Usually these are I2C devices, but not necessarily. In order to provide the driver with a consistent interface to these sub-devices the v4l2_subdev struct (v4l2-subdev.h) was created.

Each sub-device driver must have a v4l2_subdev struct. This struct can be stand-alone for simple sub-devices or it might be embedded in a larger struct if more state information needs to be stored. Usually there is a low-level device struct (e.g. i2c_client) that contains the device data as setup by the kernel. It is recommended to store that pointer in the private data of v4l2_subdev using v4l2_set_subdevdata(). That makes it easy to go from a v4l2_subdev to the actual low-level bus-specific device data.

You also need a way to go from the low-level struct to v4l2_subdev. For the common i2c_client struct the i2c_set_clientdata() call is used to store a v4l2_subdev pointer, for other buses you may have to use other methods.

Bridges might also need to store per-subdev private data, such as a pointer to bridge-specific per-subdev private data. The v4l2_subdev structure provides host private data for that purpose that can be accessed with v4l2_get_subdev_hostdata() and v4l2_set_subdev_hostdata().

From the bridge driver perspective, you load the sub-device module and somehow obtain the v4l2_subdev pointer. For i2c devices this is easy: you call i2c_get_clientdata(). For other buses something similar needs to be done. Helper functions exists for sub-devices on an I2C bus that do most of this tricky work for you.

Each v4l2_subdev contains function pointers that sub-device drivers can implement (or leave NULL if it is not applicable). Since sub-devices can do so many different things and you do not want to end up with a huge ops struct of which only a handful of ops are commonly implemented, the function pointers are sorted according to category and each category has its own ops struct.

The top-level ops struct contains pointers to the category ops structs, which may be NULL if the subdev driver does not support anything from that category.

It looks like this:

```
struct v4l2_subdev_core_ops {
        int (*log_status)(struct v4l2_subdev *sd);
        int (*init)(struct v4l2_subdev *sd, u32 val);
        ...
};
```

(continues on next page)

(continued from previous page)

```
struct v4l2_subdev_tuner_ops {
        ...
};

struct v4l2_subdev_audio_ops {
        ...
};

struct v4l2_subdev_video_ops {
        ...
};

struct v4l2_subdev_pad_ops {
        ...
};

struct v4l2_subdev_ops {
        const struct v4l2_subdev_core_ops  *core;
        const struct v4l2_subdev_tuner_ops *tuner;
        const struct v4l2_subdev_audio_ops *audio;
        const struct v4l2_subdev_video_ops *video;
        const struct v4l2_subdev_pad_ops *video;
};
```

The core ops are common to all subdevs, the other categories are implemented depending on the sub-device. E.g. a video device is unlikely to support the audio ops and vice versa.

This setup limits the number of function pointers while still making it easy to add new ops and categories.

A sub-device driver initializes the `v4l2_subdev` struct using:

`v4l2_subdev_init (sd, &ops)`.

Afterwards you need to initialize `sd->name` with a unique name and set the module owner. This is done for you if you use the i2c helper functions.

If integration with the media framework is needed, you must initialize the `media_entity` struct embedded in the `v4l2_subdev` struct (entity field) by calling `media_entity_pads_init()`, if the entity has pads:

```
struct media_pad *pads = &my_sd->pads;
int err;

err = media_entity_pads_init(&sd->entity, npads, pads);
```

The pads array must have been previously initialized. There is no need to manually set the struct `media_entity` function and name fields, but the revision field must be initialized if needed.

A reference to the entity will be automatically acquired/released when the subdev device node (if any) is opened/closed.

Don’t forget to cleanup the media entity before the sub-device is destroyed:

```
media_entity_cleanup(&sd->entity);
```

If the subdev driver intends to process video and integrate with the media framework, it must implement format related functionality using `v4l2_subdev_pad_ops` instead of `v4l2_subdev_video_ops`.

In that case, the subdev driver may set the link_validate field to provide its own link validation function. The link validation function is called for every link in the pipeline where both of the ends of the links are V4L2 sub-devices. The driver is still responsible for validating the correctness of the format configuration between sub-devices and video nodes.

If link_validate op is not set, the default function `v4l2_subdev_link_validate_default()` is used instead. This function ensures that width, height and the media bus pixel code are equal on both source and sink of the link. Subdev drivers are also free to use this function to perform the checks mentioned above in addition to their own checks.

There are currently two ways to register subdevices with the V4L2 core. The first (traditional) possibility is to have subdevices registered by bridge drivers. This can be done when the bridge driver has the complete information about subdevices connected to it and knows exactly when to register them. This is typically the case for internal subdevices, like video data processing units within SoCs or complex PCI(e) boards, camera sensors in USB cameras or connected to SoCs, which pass information about them to bridge drivers, usually in their platform data.

There are however also situations where subdevices have to be registered asynchronously to bridge devices. An example of such a configuration is a Device Tree based system where information about subdevices is made available to the system independently from the bridge devices, e.g. when subdevices are defined in DT as I2C device nodes. The API used in this second case is described further below.

Using one or the other registration method only affects the probing process, the run-time bridge-subdevice interaction is in both cases the same.

In the synchronous case a device (bridge) driver needs to register the `v4l2_subdev` with the v4l2_device:

`v4l2_device_register_subdev` (`v4l2_dev`, `sd`).

This can fail if the subdev module disappeared before it could be registered. After this function was called successfully the subdev->dev field points to the `v4l2_device`.

If the v4l2_device parent device has a non-NULL mdev field, the sub-device entity will be automatically registered with the media device.

You can unregister a sub-device using:

`v4l2_device_unregister_subdev` (`sd`).

Afterwards the subdev module can be unloaded and `sd->dev == NULL`.

You can call an ops function either directly:

```
err = sd->ops->core->g_std(sd, &norm);
```

but it is better and easier to use this macro:

```
err = v4l2_subdev_call(sd, core, g_std, &norm);
```

The macro will to the right NULL pointer checks and returns -ENODEV if sd is NULL, -ENOIOCTLCMD if either sd->core or sd->core->g_std is NULL, or the actual result of the sd->ops->core->g_std ops.

It is also possible to call all or a subset of the sub-devices:

```
v4l2_device_call_all(v4l2_dev, 0, core, g_std, &norm);
```

Any subdev that does not support this ops is skipped and error results are ignored. If you want to check for errors use this:

```
err = v4l2_device_call_until_err(v4l2_dev, 0, core, g_std, &norm);
```

Any error except -ENOIOCTLCMD will exit the loop with that error. If no errors (except -ENOIOCTLCMD) occurred, then 0 is returned.

The second argument to both calls is a group ID. If 0, then all subdevs are called. If non-zero, then only those whose group ID match that value will be called. Before a bridge driver registers a subdev it can set sd->grp_id to whatever value it wants (it's 0 by default). This value is owned by the bridge driver and the sub-device driver will never modify or use it.

The group ID gives the bridge driver more control how callbacks are called. For example, there may be multiple audio chips on a board, each capable of changing the volume. But usually only one will actually be used when the user want to change the volume. You can set the group ID for that subdev to e.g. AUDIO_CONTROLLER and specify that as the group ID value when calling v4l2_device_call_all(). That ensures that it will only go to the subdev that needs it.

If the sub-device needs to notify its v4l2_device parent of an event, then it can call v4l2_subdev_notify(sd, notification, arg). This macro checks whether there is a notify() callback defined and returns -ENODEV if not. Otherwise the result of the notify() call is returned.

The advantage of using v4l2_subdev is that it is a generic struct and does not contain any knowledge about the underlying hardware. So a driver might contain several subdevs that use an I2C bus, but also a subdev that is controlled through GPIO pins. This distinction is only relevant when setting up the device, but once the subdev is registered it is completely transparent.

In the asynchronous case subdevice probing can be invoked independently of the bridge driver availability. The subdevice driver then has to verify whether all the requirements for a successful probing are satisfied. This can include a check for a master clock availability. If any of the conditions aren't satisfied the driver might decide to return -EPROBE_DEFER to request further reprobing attempts. Once all conditions are met the subdevice shall be registered using the v4l2_async_register_subdev() function. Unregistration is performed using the v4l2_async_unregister_subdev() call. Subdevices registered this way are stored in a global list of subdevices, ready to be picked up by bridge drivers.

Bridge drivers in turn have to register a notifier object. This is performed using the v4l2_async_notifier_register() call. To unregister the notifier the driver has to call v4l2_async_notifier_unregister(). The former of the two functions

takes two arguments: a pointer to struct v4l2_device and a pointer to struct v4l2_async_notifier.

Before registering the notifier, bridge drivers must do two things: first, the notifier must be initialized using the v4l2_async_notifier_init(). Second, bridge drivers can then begin to form a list of subdevice descriptors that the bridge device needs for its operation. Subdevice descriptors are added to the notifier using the v4l2_async_notifier_add_subdev() call. This function takes two arguments: a pointer to struct v4l2_async_notifier, and a pointer to the subdevice descripter, which is of type struct v4l2_async_subdev.

The V4L2 core will then use these descriptors to match asynchronously registered subdevices to them. If a match is detected the .bound() notifier callback is called. After all subdevices have been located the .complete() callback is called. When a subdevice is removed from the system the .unbind() method is called. All three callbacks are optional.

53.1.8 V4L2 sub-device userspace API

Bridge drivers traditionally expose one or multiple video nodes to userspace, and control subdevices through the v4l2_subdev_ops operations in response to video node operations. This hides the complexity of the underlying hardware from applications. For complex devices, finer-grained control of the device than what the video nodes offer may be required. In those cases, bridge drivers that implement the media controller API may opt for making the subdevice operations directly accessible from userpace.

Device nodes named v4l-subdevX can be created in /dev to access sub-devices directly. If a sub-device supports direct userspace configuration it must set the V4L2_SUBDEV_FL_HAS_DEVNODE flag before being registered.

After registering sub-devices, the v4l2_device driver can create device nodes for all registered sub-devices marked with V4L2_SUBDEV_FL_HAS_DEVNODE by calling v4l2_device_register_subdev_nodes(). Those device nodes will be automatically removed when sub-devices are unregistered.

The device node handles a subset of the V4L2 API.

VIDIOC_QUERYCTRL, VIDIOC_QUERYMENU, VIDIOC_G_CTRL, VIDIOC_S_CTRL, VIDIOC_G_EXT_CTRLS, VIDIOC_S_EXT_CTRLS and VIDIOC_TRY_EXT_CTRLS:

> The controls ioctls are identical to the ones defined in V4L2. They behave identically, with the only exception that they deal only with controls implemented in the sub-device. Depending on the driver, those controls can be also be accessed through one (or several) V4L2 device nodes.

VIDIOC_DQEVENT, VIDIOC_SUBSCRIBE_EVENT and VIDIOC_UNSUBSCRIBE_EVENT

> The events ioctls are identical to the ones defined in V4L2. They behave identically, with the only exception that they deal only with events generated by the sub-device. Depending on the driver, those events can also be reported by one (or several) V4L2 device nodes.
>
> Sub-device drivers that want to use events need to set the V4L2_SUBDEV_FL_HAS_EVENTS v4l2_subdev.flags before registering the

sub-device. After registration events can be queued as usual on the v4l2_subdev.devnode device node.

To properly support events, the poll() file operation is also implemented.

Private ioctls

All ioctls not in the above list are passed directly to the sub-device driver through the core::ioctl operation.

53.1.9 Read-only sub-device userspace API

Bridge drivers that control their connected subdevices through direct calls to the kernel API realized by v4l2_subdev_ops structure do not usually want userspace to be able to change the same parameters through the subdevice device node and thus do not usually register any.

It is sometimes useful to report to userspace the current subdevice configuration through a read-only API, that does not permit applications to change to the device parameters but allows interfacing to the subdevice device node to inspect them.

For instance, to implement cameras based on computational photography, userspace needs to know the detailed camera sensor configuration (in terms of skipping, binning, cropping and scaling) for each supported output resolution. To support such use cases, bridge drivers may expose the subdevice operations to userspace through a read-only API.

To create a read-only device node for all the subdevices registered with the V4L2_SUBDEV_FL_HAS_DEVNODE set, the v4l2_device driver should call v4l2_device_register_ro_subdev_nodes().

Access to the following ioctls for userspace applications is restricted on sub-device device nodes registered with v4l2_device_register_ro_subdev_nodes().

VIDIOC_SUBDEV_S_FMT, VIDIOC_SUBDEV_S_CROP, VIDIOC_SUBDEV_S_SELECTION:

These ioctls are only allowed on a read-only subdevice device node for the V4L2_SUBDEV_FORMAT_TRY formats and selection rectangles.

VIDIOC_SUBDEV_S_FRAME_INTERVAL, VIDIOC_SUBDEV_S_DV_TIMINGS, VIDIOC_SUBDEV_S_STD:

These ioctls are not allowed on a read-only subdevice node.

In case the ioctl is not allowed, or the format to modify is set to V4L2_SUBDEV_FORMAT_ACTIVE, the core returns a negative error code and the errno variable is set to -EPERM.

53.1.10 I2C sub-device drivers

Since these drivers are so common, special helper functions are available to ease the use of these drivers (v4l2-common.h).

The recommended method of adding v4l2_subdev support to an I2C driver is to embed the v4l2_subdev struct into the state struct that is created for each I2C device instance. Very simple devices have no state struct and in that case you can just create a v4l2_subdev directly.

A typical state struct would look like this (where 'chipname' is replaced by the name of the chip):

```
struct chipname_state {
        struct v4l2_subdev sd;
        ...  /* additional state fields */
};
```

Initialize the v4l2_subdev struct as follows:

```
v4l2_i2c_subdev_init(&state->sd, client, subdev_ops);
```

This function will fill in all the fields of v4l2_subdev ensure that the v4l2_subdev and i2c_client both point to one another.

You should also add a helper inline function to go from a v4l2_subdev pointer to a chipname_state struct:

```
static inline struct chipname_state *to_state(struct v4l2_subdev *sd)
{
        return container_of(sd, struct chipname_state, sd);
}
```

Use this to go from the v4l2_subdev struct to the i2c_client struct:

```
struct i2c_client *client = v4l2_get_subdevdata(sd);
```

And this to go from an i2c_client to a v4l2_subdev struct:

```
struct v4l2_subdev *sd = i2c_get_clientdata(client);
```

Make sure to call v4l2_device_unregister_subdev(sd) when the remove() callback is called. This will unregister the sub-device from the bridge driver. It is safe to call this even if the sub-device was never registered.

You need to do this because when the bridge driver destroys the i2c adapter the remove() callbacks are called of the i2c devices on that adapter. After that the corresponding v4l2_subdev structures are invalid, so they have to be unregistered first. Calling v4l2_device_unregister_subdev(sd) from the remove() callback ensures that this is always done correctly.

The bridge driver also has some helper functions it can use:

```
struct v4l2_subdev *sd = v4l2_i2c_new_subdev(v4l2_dev, adapter,
                            "module_foo", "chipid", 0x36, NULL);
```

This loads the given module (can be NULL if no module needs to be loaded) and calls i2c_new_client_device() with the given i2c_adapter and chip/address arguments. If all goes well, then it registers the subdev with the v4l2_device.

You can also use the last argument of v4l2_i2c_new_subdev() to pass an array of possible I2C addresses that it should probe. These probe addresses are only used if the previous argument is 0. A non-zero argument means that you know the exact i2c address so in that case no probing will take place.

Both functions return NULL if something went wrong.

Note that the chipid you pass to v4l2_i2c_new_subdev() is usually the same as the module name. It allows you to specify a chip variant, e.g. “saa7114” or “saa7115” . In general though the i2c driver autodetects this. The use of chipid is something that needs to be looked at more closely at a later date. It differs between i2c drivers and as such can be confusing. To see which chip variants are supported you can look in the i2c driver code for the i2c_device_id table. This lists all the possibilities.

There are one more helper function:

v4l2_i2c_new_subdev_board() uses an i2c_board_info struct which is passed to the i2c driver and replaces the irq, platform_data and addr arguments.

If the subdev supports the s_config core ops, then that op is called with the irq and platform_data arguments after the subdev was setup.

The v4l2_i2c_new_subdev() function will call v4l2_i2c_new_subdev_board(), internally filling a i2c_board_info structure using the client_type and the addr to fill it.

53.1.11 V4L2 sub-device functions and data structures

struct **v4l2_decode_vbi_line**
used to decode_vbi_line

Definition

```
struct v4l2_decode_vbi_line {
  u32 is_second_field;
  u8 *p;
  u32 line;
  u32 type;
};
```

Members

is_second_field Set to 0 for the first (odd) field; set to 1 for the second (even) field.

p Pointer to the sliced VBI data from the decoder. On exit, points to the start of the payload.

line Line number of the sliced VBI data (1-23)

type VBI service type (V4L2_SLICED_*). 0 if no service found

enum **v4l2_subdev_io_pin_bits**
Subdevice external IO pin configuration bits

Constants

V4L2_SUBDEV_IO_PIN_DISABLE disables a pin config. ENABLE assumed.

V4L2_SUBDEV_IO_PIN_OUTPUT set it if pin is an output.

V4L2_SUBDEV_IO_PIN_INPUT set it if pin is an input.

V4L2_SUBDEV_IO_PIN_SET_VALUE to set the output value via `struct v4l2_subdev_io_pin_config->value`.

V4L2_SUBDEV_IO_PIN_ACTIVE_LOW pin active is bit 0. Otherwise, ACTIVE HIGH is assumed.

struct **v4l2_subdev_io_pin_config**
Subdevice external IO pin configuration

Definition

```
struct v4l2_subdev_io_pin_config {
  u32 flags;
  u8 pin;
  u8 function;
  u8 value;
  u8 strength;
};
```

Members

flags bitmask with flags for this pin's config, whose bits are defined by `enum v4l2_subdev_io_pin_bits`.

pin Chip external IO pin to configure

function Internal signal pad/function to route to IO pin

value Initial value for pin - e.g. GPIO output value

strength Pin drive strength

struct **v4l2_subdev_core_ops**
Define core ops callbacks for subdevs

Definition

```
struct v4l2_subdev_core_ops {
  int (*log_status)(struct v4l2_subdev *sd);
  int (*s_io_pin_config)(struct v4l2_subdev *sd, size_t n, struct v4l2_
↪subdev_io_pin_config *pincfg);
  int (*init)(struct v4l2_subdev *sd, u32 val);
  int (*load_fw)(struct v4l2_subdev *sd);
  int (*reset)(struct v4l2_subdev *sd, u32 val);
  int (*s_gpio)(struct v4l2_subdev *sd, u32 val);
  long (*ioctl)(struct v4l2_subdev *sd, unsigned int cmd, void *arg);
#ifdef CONFIG_COMPAT;
  long (*compat_ioctl32)(struct v4l2_subdev *sd, unsigned int cmd,␣
↪unsigned long arg);
#endif;
```

(continues on next page)

(continued from previous page)

```
#ifdef CONFIG_VIDEO_ADV_DEBUG;
  int (*g_register)(struct v4l2_subdev *sd, struct v4l2_dbg_register *reg);
  int (*s_register)(struct v4l2_subdev *sd, const struct v4l2_dbg_register␣
↪*reg);
#endif;
  int (*s_power)(struct v4l2_subdev *sd, int on);
  int (*interrupt_service_routine)(struct v4l2_subdev *sd, u32 status,␣
↪bool *handled);
  int (*subscribe_event)(struct v4l2_subdev *sd, struct v4l2_fh *fh,␣
↪struct v4l2_event_subscription *sub);
  int (*unsubscribe_event)(struct v4l2_subdev *sd, struct v4l2_fh *fh,␣
↪struct v4l2_event_subscription *sub);
};
```

Members

log_status callback for VIDIOC_LOG_STATUS() ioctl handler code.

s_io_pin_config configure one or more chip I/O pins for chips that multiplex different internal signal pads out to IO pins. This function takes a pointer to an array of ‘n’ pin configuration entries, one for each pin being configured. This function could be called at times other than just subdevice initialization.

init initialize the sensor registers to some sort of reasonable default values. Do not use for new drivers and should be removed in existing drivers.

load_fw load firmware.

reset generic reset command. The argument selects which subsystems to reset. Passing 0 will always reset the whole chip. Do not use for new drivers without discussing this first on the linux-media mailinglist. There should be no reason normally to reset a device.

s_gpio set GPIO pins. Very simple right now, might need to be extended with a direction argument if needed.

ioctl called at the end of ioctl() syscall handler at the V4L2 core. used to provide support for private ioctls used on the driver.

compat_ioctl32 called when a 32 bits application uses a 64 bits Kernel, in order to fix data passed from/to userspace.

g_register callback for VIDIOC_DBG_G_REGISTER() ioctl handler code.

s_register callback for VIDIOC_DBG_S_REGISTER() ioctl handler code.

s_power puts subdevice in power saving mode (on == 0) or normal operation mode (on == 1).

interrupt_service_routine Called by the bridge chip’s interrupt service handler, when an interrupt status has be raised due to this subdev, so that this subdev can handle the details. It may schedule work to be performed later. It must not sleep. **Called from an IRQ context**.

subscribe_event used by the drivers to request the control framework that for it to be warned when the value of a control changes.

unsubscribe_event remove event subscription from the control framework.

struct **v4l2_subdev_tuner_ops**
Callbacks used when v4l device was opened in radio mode.

Definition

```
struct v4l2_subdev_tuner_ops {
  int (*standby)(struct v4l2_subdev *sd);
  int (*s_radio)(struct v4l2_subdev *sd);
  int (*s_frequency)(struct v4l2_subdev *sd, const struct v4l2_frequency␣
↪*freq);
  int (*g_frequency)(struct v4l2_subdev *sd, struct v4l2_frequency *freq);
  int (*enum_freq_bands)(struct v4l2_subdev *sd, struct v4l2_frequency_
↪band *band);
  int (*g_tuner)(struct v4l2_subdev *sd, struct v4l2_tuner *vt);
  int (*s_tuner)(struct v4l2_subdev *sd, const struct v4l2_tuner *vt);
  int (*g_modulator)(struct v4l2_subdev *sd, struct v4l2_modulator *vm);
  int (*s_modulator)(struct v4l2_subdev *sd, const struct v4l2_modulator␣
↪*vm);
  int (*s_type_addr)(struct v4l2_subdev *sd, struct tuner_setup *type);
  int (*s_config)(struct v4l2_subdev *sd, const struct v4l2_priv_tun_
↪config *config);
};
```

Members

standby puts the tuner in standby mode. It will be woken up automatically the next time it is used.

s_radio callback that switches the tuner to radio mode. drivers should explicitly call it when a tuner ops should operate on radio mode, before being able to handle it. Used on devices that have both AM/FM radio receiver and TV.

s_frequency callback for VIDIOC_S_FREQUENCY() ioctl handler code.

g_frequency callback for VIDIOC_G_FREQUENCY() ioctl handler code. freq->type must be filled in. Normally done by `video_ioctl2()` or the bridge driver.

enum_freq_bands callback for VIDIOC_ENUM_FREQ_BANDS() ioctl handler code.

g_tuner callback for VIDIOC_G_TUNER() ioctl handler code.

s_tuner callback for VIDIOC_S_TUNER() ioctl handler code. **vt->type** must be filled in. Normally done by video_ioctl2 or the bridge driver.

g_modulator callback for VIDIOC_G_MODULATOR() ioctl handler code.

s_modulator callback for VIDIOC_S_MODULATOR() ioctl handler code.

s_type_addr sets tuner type and its I2C addr.

s_config sets tda9887 specific stuff, like port1, port2 and qss

Description

Note: On devices that have both AM/FM and TV, it is up to the driver to explicitly call s_radio when the tuner should be switched to radio mode, before handling

other `struct v4l2_subdev_tuner_ops` that would require it. An example of such usage is:

```
static void s_frequency(void *priv, const struct v4l2_frequency *f)
{
      ...
      if (f.type == V4L2_TUNER_RADIO)
              v4l2_device_call_all(v4l2_dev, 0, tuner, s_radio);
      ...
      v4l2_device_call_all(v4l2_dev, 0, tuner, s_frequency);
}
```

---

struct **v4l2_subdev_audio_ops**
Callbacks used for audio-related settings

Definition

```
struct v4l2_subdev_audio_ops {
  int (*s_clock_freq)(struct v4l2_subdev *sd, u32 freq);
  int (*s_i2s_clock_freq)(struct v4l2_subdev *sd, u32 freq);
  int (*s_routing)(struct v4l2_subdev *sd, u32 input, u32 output, u32␣
↪config);
  int (*s_stream)(struct v4l2_subdev *sd, int enable);
};
```

Members

s_clock_freq set the frequency (in Hz) of the audio clock output. Used to slave an audio processor to the video decoder, ensuring that audio and video remain synchronized. Usual values for the frequency are 48000, 44100 or 32000 Hz. If the frequency is not supported, then -EINVAL is returned.

s_i2s_clock_freq sets I2S speed in bps. This is used to provide a standard way to select I2S clock used by driving digital audio streams at some board designs. Usual values for the frequency are 1024000 and 2048000. If the frequency is not supported, then `-EINVAL` is returned.

s_routing used to define the input and/or output pins of an audio chip, and any additional configuration data. Never attempt to use user-level input IDs (e.g. Composite, S-Video, Tuner) at this level. An i2c device shouldn't know about whether an input pin is connected to a Composite connector, become on another board or platform it might be connected to something else entirely. The calling driver is responsible for mapping a user-level input to the right pins on the i2c device.

s_stream used to notify the audio code that stream will start or has stopped.

enum **v4l2_mbus_frame_desc_flags**
media bus frame description flags

Constants

V4L2_MBUS_FRAME_DESC_FL_LEN_MAX Indicates that `struct v4l2_mbus_frame_desc_entry->length` field specifies maximum data length.

V4L2_MBUS_FRAME_DESC_FL_BLOB Indicates that the format does not have line offsets, i.e. the receiver should use 1D DMA.

struct **v4l2_mbus_frame_desc_entry**
media bus frame description structure

Definition

```
struct v4l2_mbus_frame_desc_entry {
  enum v4l2_mbus_frame_desc_flags flags;
  u32 pixelcode;
  u32 length;
};
```

Members

flags bitmask flags, as defined by `enum v4l2_mbus_frame_desc_flags`.

pixelcode media bus pixel code, valid if **flags** `FRAME_DESC_FL_BLOB` is not set.

length number of octets per frame, valid if **flags** `V4L2_MBUS_FRAME_DESC_FL_LEN_MAX` is set.

struct **v4l2_mbus_frame_desc**
media bus data frame description

Definition

```
struct v4l2_mbus_frame_desc {
  struct v4l2_mbus_frame_desc_entry entry[V4L2_FRAME_DESC_ENTRY_MAX];
  unsigned short num_entries;
};
```

Members

entry frame descriptors array

num_entries number of entries in **entry** array

struct **v4l2_subdev_video_ops**
Callbacks used when v4l device was opened in video mode.

Definition

```
struct v4l2_subdev_video_ops {
  int (*s_routing)(struct v4l2_subdev *sd, u32 input, u32 output, u32␣
↪config);
  int (*s_crystal_freq)(struct v4l2_subdev *sd, u32 freq, u32 flags);
  int (*g_std)(struct v4l2_subdev *sd, v4l2_std_id *norm);
  int (*s_std)(struct v4l2_subdev *sd, v4l2_std_id norm);
  int (*s_std_output)(struct v4l2_subdev *sd, v4l2_std_id std);
  int (*g_std_output)(struct v4l2_subdev *sd, v4l2_std_id *std);
  int (*querystd)(struct v4l2_subdev *sd, v4l2_std_id *std);
  int (*g_tvnorms)(struct v4l2_subdev *sd, v4l2_std_id *std);
  int (*g_tvnorms_output)(struct v4l2_subdev *sd, v4l2_std_id *std);
  int (*g_input_status)(struct v4l2_subdev *sd, u32 *status);
  int (*s_stream)(struct v4l2_subdev *sd, int enable);
  int (*g_pixelaspect)(struct v4l2_subdev *sd, struct v4l2_fract *aspect);
  int (*g_frame_interval)(struct v4l2_subdev *sd, struct v4l2_subdev_frame_
↪interval *interval);
  int (*s_frame_interval)(struct v4l2_subdev *sd, struct v4l2_subdev_frame_
↪interval *interval);
```

(continues on next page)

(continued from previous page)

```
  int (*s_dv_timings)(struct v4l2_subdev *sd, struct v4l2_dv_timings␣
↪*timings);
  int (*g_dv_timings)(struct v4l2_subdev *sd, struct v4l2_dv_timings␣
↪*timings);
  int (*query_dv_timings)(struct v4l2_subdev *sd, struct v4l2_dv_timings␣
↪*timings);
  int (*g_mbus_config)(struct v4l2_subdev *sd, struct v4l2_mbus_config␣
↪*cfg);
  int (*s_mbus_config)(struct v4l2_subdev *sd, const struct v4l2_mbus_
↪config *cfg);
  int (*s_rx_buffer)(struct v4l2_subdev *sd, void *buf, unsigned int␣
↪*size);
};
```

Members

s_routing see s_routing in audio_ops, except this version is for video devices.

s_crystal_freq sets the frequency of the crystal used to generate the clocks in Hz. An extra flags field allows device specific configuration regarding clock frequency dividers, etc. If not used, then set flags to 0. If the frequency is not supported, then -EINVAL is returned.

g_std callback for VIDIOC_G_STD() ioctl handler code.

s_std callback for VIDIOC_S_STD() ioctl handler code.

s_std_output set v4l2_std_id for video OUTPUT devices. This is ignored by video input devices.

g_std_output get current standard for video OUTPUT devices. This is ignored by video input devices.

querystd callback for VIDIOC_QUERYSTD() ioctl handler code.

g_tvnorms get `v4l2_std_id` with all standards supported by the video CAPTURE device. This is ignored by video output devices.

g_tvnorms_output get v4l2_std_id with all standards supported by the video OUTPUT device. This is ignored by video capture devices.

g_input_status get input status. Same as the status field in the `struct v4l2_input`

s_stream used to notify the driver that a video stream will start or has stopped.

g_pixelaspect callback to return the pixelaspect ratio.

g_frame_interval callback for VIDIOC_SUBDEV_G_FRAME_INTERVAL() ioctl handler code.

s_frame_interval callback for VIDIOC_SUBDEV_S_FRAME_INTERVAL() ioctl handler code.

s_dv_timings Set custom dv timings in the sub device. This is used when sub device is capable of setting detailed timing information in the hardware to generate/detect the video signal.

g_dv_timings Get custom dv timings in the sub device.

query_dv_timings callback for VIDIOC_QUERY_DV_TIMINGS() ioctl handler code.

g_mbus_config get supported mediabus configurations

s_mbus_config set a certain mediabus configuration. This operation is added for compatibility with soc-camera drivers and should not be used by new software.

s_rx_buffer set a host allocated memory buffer for the subdev. The subdev can adjust **size** to a lower value and must not write more data to the buffer starting at **data** than the original value of **size**.

struct **v4l2_subdev_vbi_ops**
Callbacks used when v4l device was opened in video mode via the vbi device node.

Definition

```
struct v4l2_subdev_vbi_ops {
  int (*decode_vbi_line)(struct v4l2_subdev *sd, struct v4l2_decode_vbi_
↪line *vbi_line);
  int (*s_vbi_data)(struct v4l2_subdev *sd, const struct v4l2_sliced_vbi_
↪data *vbi_data);
  int (*g_vbi_data)(struct v4l2_subdev *sd, struct v4l2_sliced_vbi_data␣
↪*vbi_data);
  int (*g_sliced_vbi_cap)(struct v4l2_subdev *sd, struct v4l2_sliced_vbi_
↪cap *cap);
  int (*s_raw_fmt)(struct v4l2_subdev *sd, struct v4l2_vbi_format *fmt);
  int (*g_sliced_fmt)(struct v4l2_subdev *sd, struct v4l2_sliced_vbi_
↪format *fmt);
  int (*s_sliced_fmt)(struct v4l2_subdev *sd, struct v4l2_sliced_vbi_
↪format *fmt);
};
```

Members

decode_vbi_line video decoders that support sliced VBI need to implement this ioctl. Field p of the `struct v4l2_decode_vbi_line` is set to the start of the VBI data that was generated by the decoder. The driver then parses the sliced VBI data and sets the other fields in the struct accordingly. The pointer p is updated to point to the start of the payload which can be copied verbatim into the data field of the `struct v4l2_sliced_vbi_data`. If no valid VBI data was found, then the type field is set to 0 on return.

s_vbi_data used to generate VBI signals on a video signal. `struct v4l2_sliced_vbi_data` is filled with the data packets that should be output. Note that if you set the line field to 0, then that VBI signal is disabled. If no valid VBI data was found, then the type field is set to 0 on return.

g_vbi_data used to obtain the sliced VBI packet from a readback register. Not all video decoders support this. If no data is available because the readback register contains invalid or erroneous data `-EIO` is returned. Note that you must fill in the 'id' member and the 'field' member (to determine whether CC data from the first or second field should be obtained).

g_sliced_vbi_cap callback for VIDIOC_G_SLICED_VBI_CAP() ioctl handler code.

s_raw_fmt setup the video encoder/decoder for raw VBI.

g_sliced_fmt retrieve the current sliced VBI settings.

s_sliced_fmt setup the sliced VBI settings.

struct **v4l2_subdev_sensor_ops**
v4l2-subdev sensor operations

Definition

```
struct v4l2_subdev_sensor_ops {
  int (*g_skip_top_lines)(struct v4l2_subdev *sd, u32 *lines);
  int (*g_skip_frames)(struct v4l2_subdev *sd, u32 *frames);
};
```

Members

g_skip_top_lines number of lines at the top of the image to be skipped. This is needed for some sensors, which always corrupt several top lines of the output image, or which send their metadata in them.

g_skip_frames number of frames to skip at stream start. This is needed for buggy sensors that generate faulty frames when they are turned on.

enum **v4l2_subdev_ir_mode**
describes the type of IR supported

Constants

V4L2_SUBDEV_IR_MODE_PULSE_WIDTH IR uses struct ir_raw_event records

struct **v4l2_subdev_ir_parameters**
Parameters for IR TX or TX

Definition

```
struct v4l2_subdev_ir_parameters {
  unsigned int bytes_per_data_element;
  enum v4l2_subdev_ir_mode mode;
  bool enable;
  bool interrupt_enable;
  bool shutdown;
  bool modulation;
  u32 max_pulse_width;
  unsigned int carrier_freq;
  unsigned int duty_cycle;
  bool invert_level;
  bool invert_carrier_sense;
  u32 noise_filter_min_width;
  unsigned int carrier_range_lower;
  unsigned int carrier_range_upper;
  u32 resolution;
};
```

Members

bytes_per_data_element bytes per data element of data in read or write call.

mode IR mode as defined by `enum v4l2_subdev_ir_mode`.

enable device is active if true

interrupt_enable IR interrupts are enabled if true

shutdown if true: set hardware to low/no power, false: normal mode

modulation if true, it uses carrier, if false: baseband

max_pulse_width maximum pulse width in ns, valid only for baseband signal

carrier_freq carrier frequency in Hz, valid only for modulated signal

duty_cycle duty cycle percentage, valid only for modulated signal

invert_level invert signal level

invert_carrier_sense Send 0/space as a carrier burst. used only in TX.

noise_filter_min_width min time of a valid pulse, in ns. Used only for RX.

carrier_range_lower Lower carrier range, in Hz, valid only for modulated signal. Used only for RX.

carrier_range_upper Upper carrier range, in Hz, valid only for modulated signal. Used only for RX.

resolution The receive resolution, in ns . Used only for RX.

struct **v4l2_subdev_ir_ops**
operations for IR subdevices

Definition

```
struct v4l2_subdev_ir_ops {
  int (*rx_read)(struct v4l2_subdev *sd, u8 *buf, size_t count, ssize_t␣
↪*num);
  int (*rx_g_parameters)(struct v4l2_subdev *sd, struct v4l2_subdev_ir_
↪parameters *params);
  int (*rx_s_parameters)(struct v4l2_subdev *sd, struct v4l2_subdev_ir_
↪parameters *params);
  int (*tx_write)(struct v4l2_subdev *sd, u8 *buf, size_t count, ssize_t␣
↪*num);
  int (*tx_g_parameters)(struct v4l2_subdev *sd, struct v4l2_subdev_ir_
↪parameters *params);
  int (*tx_s_parameters)(struct v4l2_subdev *sd, struct v4l2_subdev_ir_
↪parameters *params);
};
```

Members

rx_read Reads received codes or pulse width data. The semantics are similar to a non-blocking read() call.

rx_g_parameters Get the current operating parameters and state of the the IR receiver.

rx_s_parameters Set the current operating parameters and state of the the IR receiver. It is recommended to call [rt]x_g_parameters first to fill out the current state, and only change the fields that need to be changed. Upon return, the actual device operating parameters and state will be returned. Note that hardware limitations may prevent the actual settings from matching the requested settings - e.g. an actual carrier setting of 35,904 Hz when 36,000 Hz

was requested. An exception is when the shutdown parameter is true. The last used operational parameters will be returned, but the actual state of the hardware be different to minimize power consumption and processing when shutdown is true.

tx_write Writes codes or pulse width data for transmission. The semantics are similar to a non-blocking write() call.

tx_g_parameters Get the current operating parameters and state of the the IR transmitter.

tx_s_parameters Set the current operating parameters and state of the the IR transmitter. It is recommended to call [rt]x_g_parameters first to fill out the current state, and only change the fields that need to be changed. Upon return, the actual device operating parameters and state will be returned. Note that hardware limitations may prevent the actual settings from matching the requested settings - e.g. an actual carrier setting of 35,904 Hz when 36,000 Hz was requested. An exception is when the shutdown parameter is true. The last used operational parameters will be returned, but the actual state of the hardware be different to minimize power consumption and processing when shutdown is true.

struct **v4l2_subdev_pad_config**
Used for storing subdev pad information.

Definition

```
struct v4l2_subdev_pad_config {
  struct v4l2_mbus_framefmt try_fmt;
  struct v4l2_rect try_crop;
  struct v4l2_rect try_compose;
};
```

Members

try_fmt `struct v4l2_mbus_framefmt`

try_crop `struct v4l2_rect` to be used for crop

try_compose `struct v4l2_rect` to be used for compose

Description

This structure only needs to be passed to the pad op if the 'which' field of the main argument is set to `V4L2_SUBDEV_FORMAT_TRY`. For `V4L2_SUBDEV_FORMAT_ACTIVE` it is safe to pass `NULL`.

struct **v4l2_subdev_pad_ops**
v4l2-subdev pad level operations

Definition

```
struct v4l2_subdev_pad_ops {
  int (*init_cfg)(struct v4l2_subdev *sd, struct v4l2_subdev_pad_config␣
↪*cfg);
  int (*enum_mbus_code)(struct v4l2_subdev *sd,struct v4l2_subdev_pad_
↪config *cfg, struct v4l2_subdev_mbus_code_enum *code);
  int (*enum_frame_size)(struct v4l2_subdev *sd,struct v4l2_subdev_pad_
↪config *cfg, struct v4l2_subdev_frame_size_enum *fse);
```

(continues on next page)

(continued from previous page)

```
  int (*enum_frame_interval)(struct v4l2_subdev *sd,struct v4l2_subdev_pad_
↪config *cfg, struct v4l2_subdev_frame_interval_enum *fie);
  int (*get_fmt)(struct v4l2_subdev *sd,struct v4l2_subdev_pad_config *cfg,
↪ struct v4l2_subdev_format *format);
  int (*set_fmt)(struct v4l2_subdev *sd,struct v4l2_subdev_pad_config *cfg,
↪ struct v4l2_subdev_format *format);
  int (*get_selection)(struct v4l2_subdev *sd,struct v4l2_subdev_pad_
↪config *cfg, struct v4l2_subdev_selection *sel);
  int (*set_selection)(struct v4l2_subdev *sd,struct v4l2_subdev_pad_
↪config *cfg, struct v4l2_subdev_selection *sel);
  int (*get_edid)(struct v4l2_subdev *sd, struct v4l2_edid *edid);
  int (*set_edid)(struct v4l2_subdev *sd, struct v4l2_edid *edid);
  int (*dv_timings_cap)(struct v4l2_subdev *sd, struct v4l2_dv_timings_cap␣
↪*cap);
  int (*enum_dv_timings)(struct v4l2_subdev *sd, struct v4l2_enum_dv_
↪timings *timings);
#ifdef CONFIG_MEDIA_CONTROLLER;
  int (*link_validate)(struct v4l2_subdev *sd, struct media_link *link,
↪struct v4l2_subdev_format *source_fmt, struct v4l2_subdev_format *sink_
↪fmt);
#endif ;
  int (*get_frame_desc)(struct v4l2_subdev *sd, unsigned int pad, struct␣
↪v4l2_mbus_frame_desc *fd);
  int (*set_frame_desc)(struct v4l2_subdev *sd, unsigned int pad, struct␣
↪v4l2_mbus_frame_desc *fd);
};
```

Members

init_cfg initialize the pad config to default values

enum_mbus_code callback for VIDIOC_SUBDEV_ENUM_MBUS_CODE() ioctl handler code.

enum_frame_size callback for VIDIOC_SUBDEV_ENUM_FRAME_SIZE() ioctl handler code.

enum_frame_interval callback for VIDIOC_SUBDEV_ENUM_FRAME_INTERVAL() ioctl handler code.

get_fmt callback for VIDIOC_SUBDEV_G_FMT() ioctl handler code.

set_fmt callback for VIDIOC_SUBDEV_S_FMT() ioctl handler code.

get_selection callback for VIDIOC_SUBDEV_G_SELECTION() ioctl handler code.

set_selection callback for VIDIOC_SUBDEV_S_SELECTION() ioctl handler code.

get_edid callback for VIDIOC_SUBDEV_G_EDID() ioctl handler code.

set_edid callback for VIDIOC_SUBDEV_S_EDID() ioctl handler code.

dv_timings_cap callback for VIDIOC_SUBDEV_DV_TIMINGS_CAP() ioctl handler code.

enum_dv_timings callback for VIDIOC_SUBDEV_ENUM_DV_TIMINGS() ioctl handler code.

link_validate used by the media controller code to check if the links that belongs to a pipeline can be used for stream.

get_frame_desc get the current low level media bus frame parameters.

set_frame_desc set the low level media bus frame parameters, **fd** array may be adjusted by the subdev driver to device capabilities.

struct **v4l2_subdev_ops**
Subdev operations

Definition

```
struct v4l2_subdev_ops {
  const struct v4l2_subdev_core_ops       *core;
  const struct v4l2_subdev_tuner_ops      *tuner;
  const struct v4l2_subdev_audio_ops      *audio;
  const struct v4l2_subdev_video_ops      *video;
  const struct v4l2_subdev_vbi_ops        *vbi;
  const struct v4l2_subdev_ir_ops         *ir;
  const struct v4l2_subdev_sensor_ops     *sensor;
  const struct v4l2_subdev_pad_ops        *pad;
};
```

Members

core pointer to `struct v4l2_subdev_core_ops`. Can be `NULL`

tuner pointer to `struct v4l2_subdev_tuner_ops`. Can be `NULL`

audio pointer to `struct v4l2_subdev_audio_ops`. Can be `NULL`

video pointer to `struct v4l2_subdev_video_ops`. Can be `NULL`

vbi pointer to `struct v4l2_subdev_vbi_ops`. Can be `NULL`

ir pointer to `struct v4l2_subdev_ir_ops`. Can be `NULL`

sensor pointer to `struct v4l2_subdev_sensor_ops`. Can be `NULL`

pad pointer to `struct v4l2_subdev_pad_ops`. Can be `NULL`

struct **v4l2_subdev_internal_ops**
V4L2 subdev internal ops

Definition

```
struct v4l2_subdev_internal_ops {
  int (*registered)(struct v4l2_subdev *sd);
  void (*unregistered)(struct v4l2_subdev *sd);
  int (*open)(struct v4l2_subdev *sd, struct v4l2_subdev_fh *fh);
  int (*close)(struct v4l2_subdev *sd, struct v4l2_subdev_fh *fh);
  void (*release)(struct v4l2_subdev *sd);
};
```

Members

registered called when this subdev is registered. When called the v4l2_dev field is set to the correct v4l2_device.

unregistered called when this subdev is unregistered. When called the v4l2_dev field is still set to the correct v4l2_device.

open called when the subdev device node is opened by an application.

close called when the subdev device node is closed. Please note that it is possible for **close** to be called after **unregistered**!

release called when the last user of the subdev device is gone. This happens after the **unregistered** callback and when the last open filehandle to the v4l-subdevX device node was closed. If no device node was created for this sub-device, then the **release** callback is called right after the **unregistered** callback. The **release** callback is typically used to free the memory containing the v4l2_subdev structure. It is almost certainly required for any sub-device that sets the V4L2_SUBDEV_FL_HAS_DEVNODE flag.

Description

Note: Never call this from drivers, only the v4l2 framework can call these ops.

---

struct **v4l2_subdev_platform_data**
regulators config struct

Definition

```
struct v4l2_subdev_platform_data {
  struct regulator_bulk_data *regulators;
  int num_regulators;
  void *host_priv;
};
```

Members

regulators Optional regulators used to power on/off the subdevice

num_regulators Number of regululators

host_priv Per-subdevice data, specific for a certain video host device

struct **v4l2_subdev**
describes a V4L2 sub-device

Definition

```
struct v4l2_subdev {
#if defined(CONFIG_MEDIA_CONTROLLER);
  struct media_entity entity;
#endif;
  struct list_head list;
  struct module *owner;
  bool owner_v4l2_dev;
  u32 flags;
  struct v4l2_device *v4l2_dev;
  const struct v4l2_subdev_ops *ops;
  const struct v4l2_subdev_internal_ops *internal_ops;
  struct v4l2_ctrl_handler *ctrl_handler;
  char name[V4L2_SUBDEV_NAME_SIZE];
  u32 grp_id;
  void *dev_priv;
  void *host_priv;
```

(continues on next page)

(continued from previous page)

```
  struct video_device *devnode;
  struct device *dev;
  struct fwnode_handle *fwnode;
  struct list_head async_list;
  struct v4l2_async_subdev *asd;
  struct v4l2_async_notifier *notifier;
  struct v4l2_async_notifier *subdev_notifier;
  struct v4l2_subdev_platform_data *pdata;
};
```

Members

entity pointer to `struct media_entity`

list List of sub-devices

owner The owner is the same as the driver's `struct device` owner.

owner_v4l2_dev true if the `sd->owner` matches the owner of **v4l2_dev->dev** owner. Initialized by `v4l2_device_register_subdev()`.

flags subdev flags. Can be: `V4L2_SUBDEV_FL_IS_I2C` - Set this flag if this subdev is a i2c device; `V4L2_SUBDEV_FL_IS_SPI` - Set this flag if this subdev is a spi device; `V4L2_SUBDEV_FL_HAS_DEVNODE` - Set this flag if this subdev needs a device node; `V4L2_SUBDEV_FL_HAS_EVENTS` - Set this flag if this subdev generates events.

v4l2_dev pointer to struct `v4l2_device`

ops pointer to struct `v4l2_subdev_ops`

internal_ops pointer to struct `v4l2_subdev_internal_ops`. Never call these internal ops from within a driver!

ctrl_handler The control handler of this subdev. May be NULL.

name Name of the sub-device. Please notice that the name must be unique.

grp_id can be used to group similar subdevs. Value is driver-specific

dev_priv pointer to private data

host_priv pointer to private data used by the device where the subdev is attached.

devnode subdev device node

dev pointer to the physical device, if any

fwnode The fwnode_handle of the subdev, usually the same as either dev->of_node->fwnode or dev->fwnode (whichever is non-NULL).

async_list Links this subdev to a global subdev_list or **notifier->done** list.

asd Pointer to respective `struct v4l2_async_subdev`.

notifier Pointer to the managing notifier.

subdev_notifier A sub-device notifier implicitly registered for the sub- device using v4l2_device_register_sensor_subdev().

pdata common part of subdevice platform data

Description

Each instance of a subdev driver should create this struct, either stand-alone or embedded in a larger struct.

This structure should be initialized by `v4l2_subdev_init()` or one of its variants: `v4l2_spi_subdev_init()`, `v4l2_i2c_subdev_init()`.

media_entity_to_v4l2_subdev(ent)
Returns a `struct v4l2_subdev` from the `struct media_entity` embedded in it.

Parameters

ent pointer to `struct media_entity`.

vdev_to_v4l2_subdev(vdev)
Returns a `struct v4l2_subdev` from the `struct video_device` embedded on it.

Parameters

vdev pointer to `struct video_device`

struct **v4l2_subdev_fh**
Used for storing subdev information per file handle

Definition

```
struct v4l2_subdev_fh {
  struct v4l2_fh vfh;
  struct module *owner;
#if defined(CONFIG_VIDEO_V4L2_SUBDEV_API);
  struct v4l2_subdev_pad_config *pad;
#endif;
};
```

Members

vfh pointer to `struct v4l2_fh`

owner module pointer to the owner of this file handle

pad pointer to `struct v4l2_subdev_pad_config`

to_v4l2_subdev_fh(fh)
Returns a `struct v4l2_subdev_fh` from the `struct v4l2_fh` embedded on it.

Parameters

fh pointer to `struct v4l2_fh`

struct v4l2_mbus_framefmt * **v4l2_subdev_get_try_format**(struct v4l2_subdev * sd, struct v4l2_subdev_pad_config * cfg, unsigned int pad)

ancillary routine to call `struct v4l2_subdev_pad_config->try_fmt`

Parameters

`struct v4l2_subdev * sd` pointer to `struct v4l2_subdev`

`struct v4l2_subdev_pad_config * cfg` pointer to `struct v4l2_subdev_pad_config` array.

`unsigned int pad` index of the pad in the **cfg** array.

struct v4l2_rect * **`v4l2_subdev_get_try_crop`**(struct v4l2_subdev * sd, struct v4l2_subdev_pad_config * cfg, unsigned int pad)

ancillary routine to call `struct v4l2_subdev_pad_config->try_crop`

Parameters

`struct v4l2_subdev * sd` pointer to `struct v4l2_subdev`

`struct v4l2_subdev_pad_config * cfg` pointer to `struct v4l2_subdev_pad_config` array.

`unsigned int pad` index of the pad in the **cfg** array.

struct v4l2_rect * **`v4l2_subdev_get_try_compose`**(struct v4l2_subdev * sd, struct v4l2_subdev_pad_config * cfg, unsigned int pad)

ancillary routine to call `struct v4l2_subdev_pad_config->try_compose`

Parameters

`struct v4l2_subdev * sd` pointer to `struct v4l2_subdev`

`struct v4l2_subdev_pad_config * cfg` pointer to `struct v4l2_subdev_pad_config` array.

`unsigned int pad` index of the pad in the **cfg** array.

void **`v4l2_set_subdevdata`**(struct v4l2_subdev * sd, void * p)

Sets V4L2 dev private device data

Parameters

`struct v4l2_subdev * sd` pointer to `struct v4l2_subdev`

`void * p` pointer to the private device data to be stored.

void * **`v4l2_get_subdevdata`**(const struct v4l2_subdev * sd)

Gets V4L2 dev private device data

Parameters

`const struct v4l2_subdev * sd` pointer to `struct v4l2_subdev`

Description

Returns the pointer to the private device data to be stored.

void **`v4l2_set_subdev_hostdata`**(struct v4l2_subdev * sd, void * p)

Sets V4L2 dev private host data

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

void * p pointer to the private data to be stored.

void * **v4l2_get_subdev_hostdata**(const struct v4l2_subdev * sd)
Gets V4L2 dev private data

Parameters

const struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

Description

Returns the pointer to the private host data to be stored.

int **v4l2_subdev_get_fwnode_pad_1_to_1**(struct media_entity * entity, struct fwnode_endpoint * endpoint)
Get pad number from a subdev fwnode endpoint, assuming 1:1 port:pad

Parameters

struct media_entity * entity undescribed

struct fwnode_endpoint * endpoint undescribed

Description

entity - Pointer to the subdev entity **endpoint** - Pointer to a parsed fwnode endpoint

This function can be used as the .get_fwnode_pad operation for subdevices that map port numbers and pad indexes 1:1. If the endpoint is owned by the subdevice, the function returns the endpoint port number.

Returns the endpoint port number on success or a negative error code.

int **v4l2_subdev_link_validate_default**(struct v4l2_subdev * sd, struct media_link * link, struct v4l2_subdev_format * source_fmt, struct v4l2_subdev_format * sink_fmt)
validates a media link

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

struct media_link * link pointer to `struct media_link`

struct v4l2_subdev_format * source_fmt pointer to `struct v4l2_subdev_format`

struct v4l2_subdev_format * sink_fmt pointer to `struct v4l2_subdev_format`

Description

This function ensures that width, height and the media bus pixel code are equal on both source and sink of the link.

int **v4l2_subdev_link_validate**(struct media_link * link)
validates a media link

Parameters

struct media_link * link pointer to `struct media_link`

Description

This function calls the subdev' s link_validate ops to validate if a media link is valid for streaming. It also internally calls `v4l2_subdev_link_validate_default()` to ensure that width, height and the media bus pixel code are equal on both source and sink of the link.

struct v4l2_subdev_pad_config * **v4l2_subdev_alloc_pad_config**(struct v4l2_subdev * sd)
Allocates memory for pad config

Parameters

struct v4l2_subdev * sd pointer to struct v4l2_subdev

void **v4l2_subdev_free_pad_config**(struct v4l2_subdev_pad_config * cfg)
Frees memory allocated by `v4l2_subdev_alloc_pad_config()`.

Parameters

struct v4l2_subdev_pad_config * cfg pointer to `struct v4l2_subdev_pad_config`

void **v4l2_subdev_init**(struct v4l2_subdev * sd, const struct v4l2_subdev_ops * ops)
initializes the sub-device struct

Parameters

struct v4l2_subdev * sd pointer to the `struct v4l2_subdev` to be initialized

const struct v4l2_subdev_ops * ops pointer to `struct v4l2_subdev_ops`.

v4l2_subdev_call(sd, o, f, args)
call an operation of a v4l2_subdev.

Parameters

sd pointer to the `struct v4l2_subdev`

o name of the element at `struct v4l2_subdev_ops` that contains **f**. Each element there groups a set of callbacks functions.

f callback function to be called. The callback functions are defined in groups, according to each element at `struct v4l2_subdev_ops`.

args arguments for **f**.

Example

err = v4l2_subdev_call(sd, video, s_std, norm);

v4l2_subdev_has_op(sd, o, f)
Checks if a subdev defines a certain operation.

Parameters

sd pointer to the `struct v4l2_subdev`

o The group of callback functions in `struct v4l2_subdev_ops` which **f** is a part of.

f callback function to be checked for its existence.

void **v4l2_subdev_notify_event**(struct v4l2_subdev * sd, const struct v4l2_event * ev)
Delivers event notification for subdevice

Parameters

struct v4l2_subdev * sd The subdev for which to deliver the event

const struct v4l2_event * ev The event to deliver

Description

Will deliver the specified event to all userspace event listeners which are subscribed to the v42l subdev event queue as well as to the bridge driver using the notify callback. The notification type for the notify callback will be `V4L2_DEVICE_NOTIFY_EVENT`.

enum **v4l2_async_match_type**
type of asynchronous subdevice logic to be used in order to identify a match

Constants

V4L2_ASYNC_MATCH_CUSTOM Match will use the logic provided by `struct v4l2_async_subdev.match` ops

V4L2_ASYNC_MATCH_DEVNAME Match will use the device name

V4L2_ASYNC_MATCH_I2C Match will check for I2C adapter ID and address

V4L2_ASYNC_MATCH_FWNODE Match will use firmware node

Description

This enum is used by the asyncrhronous sub-device logic to define the algorithm that will be used to match an asynchronous device.

struct **v4l2_async_subdev**
sub-device descriptor, as known to a bridge

Definition

```
struct v4l2_async_subdev {
  enum v4l2_async_match_type match_type;
  union {
    struct fwnode_handle *fwnode;
    const char *device_name;
    struct {
      int adapter_id;
      unsigned short address;
    } i2c;
    struct {
      bool (*match)(struct device *dev, struct v4l2_async_subdev *sd);
      void *priv;
    } custom;
```

(continues on next page)

(continued from previous page)

```
  } match;
  struct list_head list;
  struct list_head asd_list;
};
```

Members

match_type type of match that will be used

match union of per-bus type matching data sets

match.fwnode pointer to `struct fwnode_handle` to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_FWNODE`.

match.device_name string containing the device name to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_DEVNAME`.

match.i2c embedded struct with I2C parameters to be matched. Both **match.i2c.adapter_id** and **match.i2c.address** should be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.

match.i2c.adapter_id I2C adapter ID to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.

match.i2c.address I2C address to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.

match.custom Driver-specific match criteria. Used if **match_type** is `V4L2_ASYNC_MATCH_CUSTOM`.

match.custom.match Driver-specific match function to be used if `V4L2_ASYNC_MATCH_CUSTOM`.

match.custom.priv Driver-specific private struct with match parameters to be used if `V4L2_ASYNC_MATCH_CUSTOM`.

list used to link struct v4l2_async_subdev objects, waiting to be probed, to a notifier->waiting list

asd_list used to add struct v4l2_async_subdev objects to the master notifier **asd_list**

Description

When this struct is used as a member in a driver specific struct, the driver specific struct shall contain the `struct v4l2_async_subdev` as its first member.

struct **v4l2_async_notifier_operations**
Asynchronous V4L2 notifier operations

Definition

```
struct v4l2_async_notifier_operations {
  int (*bound)(struct v4l2_async_notifier *notifier,struct v4l2_subdev␣
↪*subdev, struct v4l2_async_subdev *asd);
  int (*complete)(struct v4l2_async_notifier *notifier);
  void (*unbind)(struct v4l2_async_notifier *notifier,struct v4l2_subdev␣
↪*subdev, struct v4l2_async_subdev *asd);
};
```

Members

bound a subdevice driver has successfully probed one of the subdevices

complete All subdevices have been probed successfully. The complete callback is only executed for the root notifier.

unbind a subdevice is leaving

struct **v4l2_async_notifier**
v4l2_device notifier data

Definition

```
struct v4l2_async_notifier {
  const struct v4l2_async_notifier_operations *ops;
  struct v4l2_device *v4l2_dev;
  struct v4l2_subdev *sd;
  struct v4l2_async_notifier *parent;
  struct list_head asd_list;
  struct list_head waiting;
  struct list_head done;
  struct list_head list;
};
```

Members

ops notifier operations

v4l2_dev v4l2_device of the root notifier, NULL otherwise

sd sub-device that registered the notifier, NULL otherwise

parent parent notifier

asd_list master list of struct v4l2_async_subdev

waiting list of struct v4l2_async_subdev, waiting for their drivers

done list of struct v4l2_subdev, already probed

list member in a global list of notifiers

void **v4l2_async_notifier_init**(struct v4l2_async_notifier * notifier)
Initialize a notifier.

Parameters

struct v4l2_async_notifier * notifier pointer to `struct v4l2_async_notifier`

Description

This function initializes the notifier **asd_list**. It must be called before the first call to **v4l2_async_notifier_add_subdev**.

int **v4l2_async_notifier_add_subdev**(struct v4l2_async_notifier * notifier,
struct v4l2_async_subdev * asd)
Add an async subdev to the notifier' s master asd list.

Parameters

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

struct v4l2_async_subdev * asd pointer to struct v4l2_async_subdev

Description

Call this function before registering a notifier to link the provided asd to the notifiers master **asd_list**.

struct v4l2_async_subdev * **v4l2_async_notifier_add_fwnode_subdev**(struct v4l2_async_notifier * notifier, struct fwnode_handle * fwnode, unsigned int asd_struct_size)

Allocate and add a fwnode async subdev to the notifier's master asd_list.

Parameters

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

struct fwnode_handle * fwnode fwnode handle of the sub-device to be matched

unsigned int asd_struct_size size of the driver's async sub-device struct, including sizeof(struct v4l2_async_subdev). The struct v4l2_async_subdev shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

Description

Allocate a fwnode-matched asd of size asd_struct_size, and add it to the notifiers **asd_list**. The function also gets a reference of the fwnode which is released later at notifier cleanup time.

int **v4l2_async_notifier_add_fwnode_remote_subdev**(struct v4l2_async_notifier * notif, struct fwnode_handle * endpoint, struct v4l2_async_subdev * asd)

Allocate and add a fwnode remote async subdev to the notifier's master asd_list.

Parameters

struct v4l2_async_notifier * notif pointer to struct v4l2_async_notifier

struct fwnode_handle * endpoint local endpoint pointing to the remote sub-device to be matched

struct v4l2_async_subdev * asd Async sub-device struct allocated by the caller. The `struct v4l2_async_subdev` shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

Description

Gets the remote endpoint of a given local endpoint, set it up for fwnode matching and adds the async sub-device to the notifier's **asd_list**. The function also gets a reference of the fwnode which is released later at notifier cleanup time.

This is just like **v4l2_async_notifier_add_fwnode_subdev**, but with the exception that the fwnode refers to a local endpoint, not the remote one, and the function relies on the caller to allocate the async sub-device struct.

struct v4l2_async_subdev * **v4l2_async_notifier_add_i2c_subdev**(struct v4l2_async_notifier * notifier, int adapter_id, unsigned short address, unsigned int asd_struct_size)

Allocate and add an i2c async subdev to the notifier's master asd_list.

Parameters

struct v4l2_async_notifier * notifier pointer to `struct v4l2_async_notifier`

int adapter_id I2C adapter ID to be matched

unsigned short address I2C address of sub-device to be matched

unsigned int asd_struct_size size of the driver's async sub-device struct, including sizeof(struct v4l2_async_subdev). The `struct v4l2_async_subdev` shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

Description

Same as above but for I2C matched sub-devices.

struct v4l2_async_subdev * **v4l2_async_notifier_add_devname_subdev**(struct v4l2_async_notifier * notifier, const char * device_name, unsigned int asd_struct_size)

Allocate and add a device-name async subdev to the notifier's master asd_list.

Parameters

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

const char * device_name device name string to be matched

unsigned int asd_struct_size size of the driver's async sub-device struct, including sizeof(struct v4l2_async_subdev). The struct v4l2_async_subdev shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

Description

Same as above but for device-name matched sub-devices.

int **v4l2_async_notifier_register**(struct v4l2_device * v4l2_dev, struct v4l2_async_notifier * notifier)
registers a subdevice asynchronous notifier

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

int **v4l2_async_subdev_notifier_register**(struct v4l2_subdev * sd, struct v4l2_async_notifier * notifier)
registers a subdevice asynchronous notifier for a sub-device

Parameters

struct v4l2_subdev * sd pointer to struct v4l2_subdev

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

void **v4l2_async_notifier_unregister**(struct v4l2_async_notifier * notifier)
unregisters a subdevice asynchronous notifier

Parameters

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

void **v4l2_async_notifier_cleanup**(struct v4l2_async_notifier * notifier)
clean up notifier resources

Parameters

struct v4l2_async_notifier * notifier the notifier the resources of which are to be cleaned up

Description

Release memory resources related to a notifier, including the async sub-devices allocated for the purposes of the notifier but not the notifier itself. The user is responsible for calling this function to clean up the notifier after calling **v4l2_async_notifier_add_subdev**, **v4l2_async_notifier_parse_fwnode_endpoints** or **v4l2_fwnode_reference_parse_sensor_c**

There is no harm from calling v4l2_async_notifier_cleanup in other cases as long as its memory has been zeroed after it has been allocated.

int **v4l2_async_register_subdev**(struct v4l2_subdev * sd)
registers a sub-device to the asynchronous subdevice framework

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

int **v4l2_async_register_subdev_sensor_common**(struct v4l2_subdev * sd)
registers a sensor sub-device to the asynchronous sub-device framework and parse set up common sensor related devices

Parameters

struct v4l2_subdev * sd pointer to struct `v4l2_subdev`

Description

This function is just like `v4l2_async_register_subdev()` with the exception that calling it will also parse firmware interfaces for remote references using `v4l2_async_notifier_parse_fwnode_sensor_common()` and registers the async sub-devices. The sub-device is similarly unregistered by calling `v4l2_async_unregister_subdev()`.

While registered, the subdev module is marked as in-use.

An error is returned if the module is no longer loaded on any attempts to register it.

void **v4l2_async_unregister_subdev**(struct v4l2_subdev * sd)
unregisters a sub-device to the asynchronous subdevice framework

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

53.1.12 V4L2 events

The V4L2 events provide a generic way to pass events to user space. The driver must use `v4l2_fh` to be able to support V4L2 events.

Events are subscribed per-filehandle. An event specification consists of a `type` and is optionally associated with an object identified through the `id` field. If unused, then the `id` is 0. So an event is uniquely identified by the `(type, id)` tuple.

The `v4l2_fh` struct has a list of subscribed events on its `subscribed` field.

When the user subscribes to an event, a `v4l2_subscribed_event` struct is added to `v4l2_fh.subscribed`, one for every subscribed event.

Each `v4l2_subscribed_event` struct ends with a `v4l2_kevent` ringbuffer, with the size given by the caller of `v4l2_event_subscribe()`. This ringbuffer is used to store any events raised by the driver.

So every `(type, ID)` event tuple will have its own `v4l2_kevent` ringbuffer. This guarantees that if a driver is generating lots of events of one type in a short time, then that will not overwrite events of another type.

But if you get more events of one type than the size of the v4l2_kevent ringbuffer, then the oldest event will be dropped and the new one added.

The v4l2_kevent struct links into the available list of the v4l2_fh struct so VIDIOC_DQEVENT will know which event to dequeue first.

Finally, if the event subscription is associated with a particular object such as a V4L2 control, then that object needs to know about that as well so that an event can be raised by that object. So the node field can be used to link the v4l2_subscribed_event struct into a list of such objects.

So to summarize:

- struct v4l2_fh has two lists: one of the subscribed events, and one of the available events.
- struct v4l2_subscribed_event has a ringbuffer of raised (pending) events of that particular type.
- If struct v4l2_subscribed_event is associated with a specific object, then that object will have an internal list of struct v4l2_subscribed_event so it knows who subscribed an event to that object.

Furthermore, the internal struct v4l2_subscribed_event has merge() and replace() callbacks which drivers can set. These callbacks are called when a new event is raised and there is no more room.

The replace() callback allows you to replace the payload of the old event with that of the new event, merging any relevant data from the old payload into the new payload that replaces it. It is called when this event type has a ringbuffer with size is one, i.e. only one event can be stored in the ringbuffer.

The merge() callback allows you to merge the oldest event payload into that of the second-oldest event payload. It is called when the ringbuffer has size is greater than one.

This way no status information is lost, just the intermediate steps leading up to that state.

A good example of these replace/merge callbacks is in v4l2-event.c: ctrls_replace() and ctrls_merge() callbacks for the control event.

Note: these callbacks can be called from interrupt context, so they must be fast.

In order to queue events to video device, drivers should call:

```
v4l2_event_queue (vdev, ev)
```

The driver’s only responsibility is to fill in the type and the data fields. The other fields will be filled in by V4L2.

Event subscription

Subscribing to an event is via:

v4l2_event_subscribe(fh, sub , elems, ops)

This function is used to implement video_device-> ioctl_ops-> vidioc_subscribe_event, but the driver must check first if the driver is able to produce events with specified event id, and then should call v4l2_event_subscribe() to subscribe the event.

The elems argument is the size of the event queue for this event. If it is 0, then the framework will fill in a default value (this depends on the event type).

The ops argument allows the driver to specify a number of callbacks:

Callback	Description
add	called when a new listener gets added (subscribing to the same event twice will only cause this callback to get called once)
del	called when a listener stops listening
replace	replace event 'old' with event 'new'.
merge	merge event 'old' into event 'new'.

All 4 callbacks are optional, if you don't want to specify any callbacks the ops argument itself maybe NULL.

Unsubscribing an event

Unsubscribing to an event is via:

v4l2_event_unsubscribe(fh, sub)

This function is used to implement video_device-> ioctl_ops-> vidioc_unsubscribe_event. A driver may call v4l2_event_unsubscribe() directly unless it wants to be involved in unsubscription process.

The special type V4L2_EVENT_ALL may be used to unsubscribe all events. The drivers may want to handle this in a special way.

Check if there's a pending event

Checking if there's a pending event is via:

v4l2_event_pending(fh)

This function returns the number of pending events. Useful when implementing poll.

How events work

Events are delivered to user space through the poll system call. The driver can use v4l2_fh->wait (a wait_queue_head_t) as the argument for poll_wait().

There are standard and private events. New standard events must use the smallest available event type. The drivers must allocate their events from their own class starting from class base. Class base is V4L2_EVENT_PRIVATE_START + n * 1000 where n is the lowest available number. The first event type in the class is reserved for future use, so the first available event type is ‘class base + 1’.

An example on how the V4L2 events may be used can be found in the OMAP 3 ISP driver (drivers/media/platform/omap3isp).

A subdev can directly send an event to the v4l2_device notify function with V4L2_DEVICE_NOTIFY_EVENT. This allows the bridge to map the subdev that sends the event to the video node(s) associated with the subdev that need to be informed about such an event.

V4L2 event functions and data structures

struct **v4l2_kevent**
Internal kernel event struct.

Definition

```
struct v4l2_kevent {
  struct list_head        list;
  struct v4l2_subscribed_event *sev;
  struct v4l2_event       event;
  u64 ts;
};
```

Members

list List node for the v4l2_fh->available list.

sev Pointer to parent v4l2_subscribed_event.

event The event itself.

ts The timestamp of the event.

struct **v4l2_subscribed_event_ops**
Subscribed event operations.

Definition

```
struct v4l2_subscribed_event_ops {
  int (*add)(struct v4l2_subscribed_event *sev, unsigned int elems);
  void (*del)(struct v4l2_subscribed_event *sev);
  void (*replace)(struct v4l2_event *old, const struct v4l2_event *new);
  void (*merge)(const struct v4l2_event *old, struct v4l2_event *new);
};
```

Members

add Optional callback, called when a new listener is added

del Optional callback, called when a listener stops listening

replace Optional callback that can replace event ‘old’ with event ‘new’.

merge Optional callback that can merge event ‘old’ into event ‘new’.

struct **v4l2_subscribed_event**
Internal struct representing a subscribed event.

Definition

```
struct v4l2_subscribed_event {
  struct list_head        list;
  u32 type;
  u32 id;
  u32 flags;
  struct v4l2_fh          *fh;
  struct list_head        node;
  const struct v4l2_subscribed_event_ops *ops;
  unsigned int            elems;
  unsigned int            first;
  unsigned int            in_use;
  struct v4l2_kevent      events[];
};
```

Members

list List node for the v4l2_fh->subscribed list.

type Event type.

id Associated object ID (e.g. control ID). 0 if there isn’t any.

flags Copy of v4l2_event_subscription->flags.

fh Filehandle that subscribed to this event.

node List node that hooks into the object’s event list (if there is one).

ops v4l2_subscribed_event_ops

elems The number of elements in the events array.

first The index of the events containing the oldest available event.

in_use The number of queued events.

events An array of **elems** events.

int **v4l2_event_dequeue**(struct v4l2_fh * fh, struct v4l2_event * event, int nonblocking)
Dequeue events from video device.

Parameters

struct v4l2_fh * fh pointer to struct v4l2_fh

struct v4l2_event * event pointer to struct v4l2_event

int nonblocking if not zero, waits for an event to arrive

void **v4l2_event_queue**(struct video_device * vdev, const struct v4l2_event * ev)
Queue events to video device.

Parameters

struct video_device * vdev pointer to `struct video_device`

const struct v4l2_event * ev pointer to `struct v4l2_event`

Description

The event will be queued for all `struct v4l2_fh` file handlers.

Note: The driver's only responsibility is to fill in the type and the data fields.The other fields will be filled in by V4L2.

---

void **v4l2_event_queue_fh**(struct v4l2_fh * fh, const struct v4l2_event * ev)
Queue events to video device.

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

const struct v4l2_event * ev pointer to `struct v4l2_event`

Description

The event will be queued only for the specified `struct v4l2_fh` file handler.

Note: The driver's only responsibility is to fill in the type and the data fields.The other fields will be filled in by V4L2.

---

int **v4l2_event_pending**(struct v4l2_fh * fh)
Check if an event is available

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

Description

Returns the number of pending events.

int **v4l2_event_subscribe**(struct v4l2_fh * fh, const struct v4l2_event_subscription * sub, unsigned int elems, const struct v4l2_subscribed_event_ops * ops)
Subscribes to an event

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

const struct v4l2_event_subscription * sub pointer to `struct v4l2_event_subscription`

unsigned int elems size of the events queue

const struct v4l2_subscribed_event_ops * ops pointer to `v4l2_subscribed_event_ops`

Description

Note: if **elems** is zero, the framework will fill in a default value, with is currently 1 element.

---

int **v4l2_event_unsubscribe**(struct v4l2_fh * fh, const struct v4l2_event_subscription * sub)
Unsubscribes to an event

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

const struct v4l2_event_subscription * sub pointer to `struct v4l2_event_subscription`

void **v4l2_event_unsubscribe_all**(struct v4l2_fh * fh)
Unsubscribes to all events

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

int **v4l2_event_subdev_unsubscribe**(struct v4l2_subdev * sd, struct v4l2_fh * fh, struct v4l2_event_subscription * sub)
Subdev variant of `v4l2_event_unsubscribe()`

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

struct v4l2_fh * fh pointer to `struct v4l2_fh`

struct v4l2_event_subscription * sub pointer to `struct v4l2_event_subscription`

Description

Note: This function should be used for the `struct v4l2_subdev_core_ops unsubscribe_event` field.

---

int **v4l2_src_change_event_subscribe**(struct v4l2_fh * fh, const struct v4l2_event_subscription * sub)
helper function that calls `v4l2_event_subscribe()` if the event is `V4L2_EVENT_SOURCE_CHANGE`.

Parameters

struct v4l2_fh * fh pointer to struct v4l2_fh

const struct v4l2_event_subscription * sub pointer to `struct v4l2_event_subscription`

int **v4l2_src_change_event_subdev_subscribe**(struct v4l2_subdev * sd, struct v4l2_fh * fh, struct v4l2_event_subscription * sub)

Variant of `v4l2_event_subscribe()`, meant to subscribe only events of the type `V4L2_EVENT_SOURCE_CHANGE`.

Parameters

`struct v4l2_subdev * sd` pointer to `struct v4l2_subdev`

`struct v4l2_fh * fh` pointer to `struct v4l2_fh`

`struct v4l2_event_subscription * sub` pointer to `struct v4l2_event_subscription`

53.1.13 V4L2 Controls

Introduction

The V4L2 control API seems simple enough, but quickly becomes very hard to implement correctly in drivers. But much of the code needed to handle controls is actually not driver specific and can be moved to the V4L core framework.

After all, the only part that a driver developer is interested in is:

1) How do I add a control?

2) How do I set the control's value? (i.e. s_ctrl)

And occasionally:

3) How do I get the control's value? (i.e. g_volatile_ctrl)

4) How do I validate the user's proposed control value? (i.e. try_ctrl)

All the rest is something that can be done centrally.

The control framework was created in order to implement all the rules of the V4L2 specification with respect to controls in a central place. And to make life as easy as possible for the driver developer.

Note that the control framework relies on the presence of a struct `v4l2_device` for V4L2 drivers and struct `v4l2_subdev` for sub-device drivers.

Objects in the framework

There are two main objects:

The `v4l2_ctrl` object describes the control properties and keeps track of the control's value (both the current value and the proposed new value).

`v4l2_ctrl_handler` is the object that keeps track of controls. It maintains a list of v4l2_ctrl objects that it owns and another list of references to controls, possibly to controls owned by other handlers.

Basic usage for V4L2 and sub-device drivers

1) Prepare the driver:

```
#include <media/v4l2-ctrls.h>
```

1.1) Add the handler to your driver's top-level struct:

For V4L2 drivers:

```
struct foo_dev {
        ...
        struct v4l2_device v4l2_dev;
        ...
        struct v4l2_ctrl_handler ctrl_handler;
        ...
};
```

For sub-device drivers:

```
struct foo_dev {
        ...
        struct v4l2_subdev sd;
        ...
        struct v4l2_ctrl_handler ctrl_handler;
        ...
};
```

1.2) Initialize the handler:

```
v4l2_ctrl_handler_init(&foo->ctrl_handler, nr_of_controls);
```

The second argument is a hint telling the function how many controls this handler is expected to handle. It will allocate a hashtable based on this information. It is a hint only.

1.3) Hook the control handler into the driver:

For V4L2 drivers:

```
foo->v4l2_dev.ctrl_handler = &foo->ctrl_handler;
```

For sub-device drivers:

```
foo->sd.ctrl_handler = &foo->ctrl_handler;
```

1.4) Clean up the handler at the end:

```
v4l2_ctrl_handler_free(&foo->ctrl_handler);
```

2) Add controls:

You add non-menu controls by calling `v4l2_ctrl_new_std()`:

```
struct v4l2_ctrl *v4l2_ctrl_new_std(struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops,
                u32 id, s32 min, s32 max, u32 step, s32 def);
```

Menu and integer menu controls are added by calling v4l2_ctrl_new_std_menu():

```
struct v4l2_ctrl *v4l2_ctrl_new_std_menu(struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops,
                u32 id, s32 max, s32 skip_mask, s32 def);
```

Menu controls with a driver specific menu are added by calling v4l2_ctrl_new_std_menu_items():

```
struct v4l2_ctrl *v4l2_ctrl_new_std_menu_items(
                struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops, u32 id, s32 max,
                s32 skip_mask, s32 def, const char * const *qmenu);
```

Standard compound controls can be added by calling v4l2_ctrl_new_std_compound():

```
struct v4l2_ctrl *v4l2_ctrl_new_std_compound(struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops, u32 id,
                const union v4l2_ctrl_ptr p_def);
```

Integer menu controls with a driver specific menu can be added by calling v4l2_ctrl_new_int_menu():

```
struct v4l2_ctrl *v4l2_ctrl_new_int_menu(struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops,
                u32 id, s32 max, s32 def, const s64 *qmenu_int);
```

These functions are typically called right after the v4l2_ctrl_handler_init():

```
static const s64 exp_bias_qmenu[] = {
       -2, -1, 0, 1, 2
};
static const char * const test_pattern[] = {
        "Disabled",
        "Vertical Bars",
        "Solid Black",
        "Solid White",
};

v4l2_ctrl_handler_init(&foo->ctrl_handler, nr_of_controls);
v4l2_ctrl_new_std(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_BRIGHTNESS, 0, 255, 1, 128);
v4l2_ctrl_new_std(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_CONTRAST, 0, 255, 1, 128);
v4l2_ctrl_new_std_menu(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_POWER_LINE_FREQUENCY,
                V4L2_CID_POWER_LINE_FREQUENCY_60HZ, 0,
                V4L2_CID_POWER_LINE_FREQUENCY_DISABLED);
v4l2_ctrl_new_int_menu(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_EXPOSURE_BIAS,
                ARRAY_SIZE(exp_bias_qmenu) - 1,
                ARRAY_SIZE(exp_bias_qmenu) / 2 - 1,
                exp_bias_qmenu);
v4l2_ctrl_new_std_menu_items(&foo->ctrl_handler, &foo_ctrl_ops,
```

(continues on next page)

(continued from previous page)

```
                V4L2_CID_TEST_PATTERN, ARRAY_SIZE(test_pattern) - 1, 0,
                0, test_pattern);
...
if (foo->ctrl_handler.error) {
        int err = foo->ctrl_handler.error;

        v4l2_ctrl_handler_free(&foo->ctrl_handler);
        return err;
}
```

The `v4l2_ctrl_new_std()` function returns the v4l2_ctrl pointer to the new control, but if you do not need to access the pointer outside the control ops, then there is no need to store it.

The `v4l2_ctrl_new_std()` function will fill in most fields based on the control ID except for the min, max, step and default values. These are passed in the last four arguments. These values are driver specific while control attributes like type, name, flags are all global. The control's current value will be set to the default value.

The `v4l2_ctrl_new_std_menu()` function is very similar but it is used for menu controls. There is no min argument since that is always 0 for menu controls, and instead of a step there is a skip_mask argument: if bit X is 1, then menu item X is skipped.

The `v4l2_ctrl_new_int_menu()` function creates a new standard integer menu control with driver-specific items in the menu. It differs from v4l2_ctrl_new_std_menu in that it doesn't have the mask argument and takes as the last argument an array of signed 64-bit integers that form an exact menu item list.

The `v4l2_ctrl_new_std_menu_items()` function is very similar to v4l2_ctrl_new_std_menu but takes an extra parameter qmenu, which is the driver specific menu for an otherwise standard menu control. A good example for this control is the test pattern control for capture/display/sensors devices that have the capability to generate test patterns. These test patterns are hardware specific, so the contents of the menu will vary from device to device.

Note that if something fails, the function will return NULL or an error and set ctrl_handler->error to the error code. If ctrl_handler->error was already set, then it will just return and do nothing. This is also true for v4l2_ctrl_handler_init if it cannot allocate the internal data structure.

This makes it easy to init the handler and just add all controls and only check the error code at the end. Saves a lot of repetitive error checking.

It is recommended to add controls in ascending control ID order: it will be a bit faster that way.

3) Optionally force initial control setup:

```
v4l2_ctrl_handler_setup(&foo->ctrl_handler);
```

This will call s_ctrl for all controls unconditionally. Effectively this initializes the hardware to the default control values. It is recommended that you do this as this

ensures that both the internal data structures and the hardware are in sync.

4) Finally: implement the `v4l2_ctrl_ops`

```
static const struct v4l2_ctrl_ops foo_ctrl_ops = {
        .s_ctrl = foo_s_ctrl,
};
```

Usually all you need is s_ctrl:

```
static int foo_s_ctrl(struct v4l2_ctrl *ctrl)
{
        struct foo *state = container_of(ctrl->handler, struct foo, ctrl_
↪handler);

        switch (ctrl->id) {
        case V4L2_CID_BRIGHTNESS:
                write_reg(0x123, ctrl->val);
                break;
        case V4L2_CID_CONTRAST:
                write_reg(0x456, ctrl->val);
                break;
        }
        return 0;
}
```

The control ops are called with the v4l2_ctrl pointer as argument. The new control value has already been validated, so all you need to do is to actually update the hardware registers.

You' re done! And this is sufficient for most of the drivers we have. No need to do any validation of control values, or implement QUERYCTRL, QUERY_EXT_CTRL and QUERYMENU. And G/S_CTRL as well as G/TRY/S_EXT_CTRLS are automatically supported.

Note: The remainder sections deal with more advanced controls topics and scenarios. In practice the basic usage as described above is sufficient for most drivers.

Inheriting Sub-device Controls

When a sub-device is registered with a V4L2 driver by calling `v4l2_device_register_subdev()` and the ctrl_handler fields of both v4l2_subdev and v4l2_device are set, then the controls of the subdev will become automatically available in the V4L2 driver as well. If the subdev driver contains controls that already exist in the V4L2 driver, then those will be skipped (so a V4L2 driver can always override a subdev control).

What happens here is that `v4l2_device_register_subdev()` calls `v4l2_ctrl_add_handler()` adding the controls of the subdev to the controls of v4l2_device.

Accessing Control Values

The following union is used inside the control framework to access control values:

```
union v4l2_ctrl_ptr {
        s32 *p_s32;
        s64 *p_s64;
        char *p_char;
        void *p;
};
```

The v4l2_ctrl struct contains these fields that can be used to access both current and new values:

```
s32 val;
struct {
        s32 val;
} cur;


union v4l2_ctrl_ptr p_new;
union v4l2_ctrl_ptr p_cur;
```

If the control has a simple s32 type type, then:

```
&ctrl->val == ctrl->p_new.p_s32
&ctrl->cur.val == ctrl->p_cur.p_s32
```

For all other types use ctrl->p_cur.p<something>. Basically the val and cur.val fields can be considered an alias since these are used so often.

Within the control ops you can freely use these. The val and cur.val speak for themselves. The p_char pointers point to character buffers of length ctrl->maximum + 1, and are always 0-terminated.

Unless the control is marked volatile the p_cur field points to the the current cached control value. When you create a new control this value is made identical to the default value. After calling `v4l2_ctrl_handler_setup()` this value is passed to the hardware. It is generally a good idea to call this function.

Whenever a new value is set that new value is automatically cached. This means that most drivers do not need to implement the g_volatile_ctrl() op. The exception is for controls that return a volatile register such as a signal strength read-out that changes continuously. In that case you will need to implement g_volatile_ctrl like this:

```
static int foo_g_volatile_ctrl(struct v4l2_ctrl *ctrl)
{
        switch (ctrl->id) {
        case V4L2_CID_BRIGHTNESS:
                ctrl->val = read_reg(0x123);
                break;
        }
}
```

Note that you use the 'new value' union as well in g_volatile_ctrl. In general controls that need to implement g_volatile_ctrl are read-only controls. If they are not,

a V4L2_EVENT_CTRL_CH_VALUE will not be generated when the control changes.

To mark a control as volatile you have to set V4L2_CTRL_FLAG_VOLATILE:

```
ctrl = v4l2_ctrl_new_std(&sd->ctrl_handler, ...);
if (ctrl)
        ctrl->flags |= V4L2_CTRL_FLAG_VOLATILE;
```

For try/s_ctrl the new values (i.e. as passed by the user) are filled in and you can modify them in try_ctrl or set them in s_ctrl. The 'cur' union contains the current value, which you can use (but not change!) as well.

If s_ctrl returns 0 (OK), then the control framework will copy the new final values to the 'cur' union.

While in g_volatile/s/try_ctrl you can access the value of all controls owned by the same handler since the handler's lock is held. If you need to access the value of controls owned by other handlers, then you have to be very careful not to introduce deadlocks.

Outside of the control ops you have to go through to helper functions to get or set a single control value safely in your driver:

```
s32 v4l2_ctrl_g_ctrl(struct v4l2_ctrl *ctrl);
int v4l2_ctrl_s_ctrl(struct v4l2_ctrl *ctrl, s32 val);
```

These functions go through the control framework just as VIDIOC_G/S_CTRL ioctls do. Don't use these inside the control ops g_volatile/s/try_ctrl, though, that will result in a deadlock since these helpers lock the handler as well.

You can also take the handler lock yourself:

```
mutex_lock(&state->ctrl_handler.lock);
pr_info("String value is '%s'\n", ctrl1->p_cur.p_char);
pr_info("Integer value is '%s'\n", ctrl2->cur.val);
mutex_unlock(&state->ctrl_handler.lock);
```

Menu Controls

The v4l2_ctrl struct contains this union:

```
union {
        u32 step;
        u32 menu_skip_mask;
};
```

For menu controls menu_skip_mask is used. What it does is that it allows you to easily exclude certain menu items. This is used in the VIDIOC_QUERYMENU implementation where you can return -EINVAL if a certain menu item is not present. Note that VIDIOC_QUERYCTRL always returns a step value of 1 for menu controls.

A good example is the MPEG Audio Layer II Bitrate menu control where the menu is a list of standardized possible bitrates. But in practice hardware implementations will only support a subset of those. By setting the skip mask you can tell the framework which menu items should be skipped. Setting it to 0 means that all menu items are supported.

You set this mask either through the v4l2_ctrl_config struct for a custom control, or by calling `v4l2_ctrl_new_std_menu()`.

Custom Controls

Driver specific controls can be created using `v4l2_ctrl_new_custom()`:

```
static const struct v4l2_ctrl_config ctrl_filter = {
        .ops = &ctrl_custom_ops,
        .id = V4L2_CID_MPEG_CX2341X_VIDEO_SPATIAL_FILTER,
        .name = "Spatial Filter",
        .type = V4L2_CTRL_TYPE_INTEGER,
        .flags = V4L2_CTRL_FLAG_SLIDER,
        .max = 15,
        .step = 1,
};

ctrl = v4l2_ctrl_new_custom(&foo->ctrl_handler, &ctrl_filter, NULL);
```

The last argument is the priv pointer which can be set to driver-specific private data.

The v4l2_ctrl_config struct also has a field to set the is_private flag.

If the name field is not set, then the framework will assume this is a standard control and will fill in the name, type and flags fields accordingly.

Active and Grabbed Controls

If you get more complex relationships between controls, then you may have to activate and deactivate controls. For example, if the Chroma AGC control is on, then the Chroma Gain control is inactive. That is, you may set it, but the value will not be used by the hardware as long as the automatic gain control is on. Typically user interfaces can disable such input fields.

You can set the 'active' status using `v4l2_ctrl_activate()`. By default all controls are active. Note that the framework does not check for this flag. It is meant purely for GUIs. The function is typically called from within s_ctrl.

The other flag is the 'grabbed' flag. A grabbed control means that you cannot change it because it is in use by some resource. Typical examples are MPEG bitrate controls that cannot be changed while capturing is in progress.

If a control is set to 'grabbed' using `v4l2_ctrl_grab()`, then the framework will return -EBUSY if an attempt is made to set this control. The `v4l2_ctrl_grab()` function is typically called from the driver when it starts or stops streaming.

Control Clusters

By default all controls are independent from the others. But in more complex scenarios you can get dependencies from one control to another. In that case you need to 'cluster' them:

```
struct foo {
        struct v4l2_ctrl_handler ctrl_handler;
#define AUDIO_CL_VOLUME (0)
#define AUDIO_CL_MUTE   (1)
        struct v4l2_ctrl *audio_cluster[2];
        ...
};

state->audio_cluster[AUDIO_CL_VOLUME] =
        v4l2_ctrl_new_std(&state->ctrl_handler, ...);
state->audio_cluster[AUDIO_CL_MUTE] =
        v4l2_ctrl_new_std(&state->ctrl_handler, ...);
v4l2_ctrl_cluster(ARRAY_SIZE(state->audio_cluster), state->audio_cluster);
```

From now on whenever one or more of the controls belonging to the same cluster is set (or 'gotten', or 'tried'), only the control ops of the first control ('volume' in this example) is called. You effectively create a new composite control. Similar to how a 'struct' works in C.

So when s_ctrl is called with V4L2_CID_AUDIO_VOLUME as argument, you should set all two controls belonging to the audio_cluster:

```
static int foo_s_ctrl(struct v4l2_ctrl *ctrl)
{
        struct foo *state = container_of(ctrl->handler, struct foo, ctrl_
↪handler);

        switch (ctrl->id) {
        case V4L2_CID_AUDIO_VOLUME: {
                struct v4l2_ctrl *mute = ctrl->cluster[AUDIO_CL_MUTE];

                write_reg(0x123, mute->val ? 0 : ctrl->val);
                break;
        }
        case V4L2_CID_CONTRAST:
                write_reg(0x456, ctrl->val);
                break;
        }
        return 0;
}
```

In the example above the following are equivalent for the VOLUME case:

```
ctrl == ctrl->cluster[AUDIO_CL_VOLUME] == state->audio_cluster[AUDIO_CL_
↪VOLUME]
ctrl->cluster[AUDIO_CL_MUTE] == state->audio_cluster[AUDIO_CL_MUTE]
```

In practice using cluster arrays like this becomes very tiresome. So instead the following equivalent method is used:

```
struct {
        /* audio cluster */
        struct v4l2_ctrl *volume;
        struct v4l2_ctrl *mute;
};
```

The anonymous struct is used to clearly 'cluster' these two control pointers, but it serves no other purpose. The effect is the same as creating an array with two control pointers. So you can just do:

```
state->volume = v4l2_ctrl_new_std(&state->ctrl_handler, ...);
state->mute = v4l2_ctrl_new_std(&state->ctrl_handler, ...);
v4l2_ctrl_cluster(2, &state->volume);
```

And in foo_s_ctrl you can use these pointers directly: state->mute->val.

Note that controls in a cluster may be NULL. For example, if for some reason mute was never added (because the hardware doesn't support that particular feature), then mute will be NULL. So in that case we have a cluster of 2 controls, of which only 1 is actually instantiated. The only restriction is that the first control of the cluster must always be present, since that is the 'master' control of the cluster. The master control is the one that identifies the cluster and that provides the pointer to the v4l2_ctrl_ops struct that is used for that cluster.

Obviously, all controls in the cluster array must be initialized to either a valid control or to NULL.

In rare cases you might want to know which controls of a cluster actually were set explicitly by the user. For this you can check the 'is_new' flag of each control. For example, in the case of a volume/mute cluster the 'is_new' flag of the mute control would be set if the user called VIDIOC_S_CTRL for mute only. If the user would call VIDIOC_S_EXT_CTRLS for both mute and volume controls, then the 'is_new' flag would be 1 for both controls.

The 'is_new' flag is always 1 when called from `v4l2_ctrl_handler_setup()`.

Handling autogain/gain-type Controls with Auto Clusters

A common type of control cluster is one that handles 'auto-foo/foo'-type controls. Typical examples are autogain/gain, autoexposure/exposure, autowhitebalance/red balance/blue balance. In all cases you have one control that determines whether another control is handled automatically by the hardware, or whether it is under manual control from the user.

If the cluster is in automatic mode, then the manual controls should be marked inactive and volatile. When the volatile controls are read the g_volatile_ctrl operation should return the value that the hardware's automatic mode set up automatically.

If the cluster is put in manual mode, then the manual controls should become active again and the volatile flag is cleared (so g_volatile_ctrl is no longer called while in manual mode). In addition just before switching to manual mode the current values as determined by the auto mode are copied as the new manual values.

Finally the V4L2_CTRL_FLAG_UPDATE should be set for the auto control since changing that control affects the control flags of the manual controls.

In order to simplify this a special variation of v4l2_ctrl_cluster was introduced:

```
void v4l2_ctrl_auto_cluster(unsigned ncontrols, struct v4l2_ctrl␣
↪**controls,
                            u8 manual_val, bool set_volatile);
```

The first two arguments are identical to v4l2_ctrl_cluster. The third argument tells the framework which value switches the cluster into manual mode. The last argument will optionally set V4L2_CTRL_FLAG_VOLATILE for the non-auto controls. If it is false, then the manual controls are never volatile. You would typically use that if the hardware does not give you the option to read back to values as determined by the auto mode (e.g. if autogain is on, the hardware doesn't allow you to obtain the current gain value).

The first control of the cluster is assumed to be the 'auto' control.

Using this function will ensure that you don't need to handle all the complex flag and volatile handling.

VIDIOC_LOG_STATUS Support

This ioctl allow you to dump the current status of a driver to the kernel log. The v4l2_ctrl_handler_log_status(ctrl_handler, prefix) can be used to dump the value of the controls owned by the given handler to the log. You can supply a prefix as well. If the prefix didn't end with a space, then ': 'will be added for you.

Different Handlers for Different Video Nodes

Usually the V4L2 driver has just one control handler that is global for all video nodes. But you can also specify different control handlers for different video nodes. You can do that by manually setting the ctrl_handler field of struct video_device.

That is no problem if there are no subdevs involved but if there are, then you need to block the automatic merging of subdev controls to the global control handler. You do that by simply setting the ctrl_handler field in struct v4l2_device to NULL. Now `v4l2_device_register_subdev()` will no longer merge subdev controls.

After each subdev was added, you will then have to call v4l2_ctrl_add_handler manually to add the subdev's control handler (sd->ctrl_handler) to the desired control handler. This control handler may be specific to the video_device or for a subset of video_device's. For example: the radio device nodes only have audio controls, while the video and vbi device nodes share the same control handler for the audio and video controls.

If you want to have one handler (e.g. for a radio device node) have a subset of another handler (e.g. for a video device node), then you should first add the controls to the first handler, add the other controls to the second handler and finally add the first handler to the second. For example:

```
v4l2_ctrl_new_std(&radio_ctrl_handler, &radio_ops, V4L2_CID_AUDIO_VOLUME, .
↪..);
v4l2_ctrl_new_std(&radio_ctrl_handler, &radio_ops, V4L2_CID_AUDIO_MUTE, ...
↪);
v4l2_ctrl_new_std(&video_ctrl_handler, &video_ops, V4L2_CID_BRIGHTNESS, ...
↪);
v4l2_ctrl_new_std(&video_ctrl_handler, &video_ops, V4L2_CID_CONTRAST, ...);
v4l2_ctrl_add_handler(&video_ctrl_handler, &radio_ctrl_handler, NULL);
```

The last argument to `v4l2_ctrl_add_handler()` is a filter function that allows you to filter which controls will be added. Set it to NULL if you want to add all controls.

Or you can add specific controls to a handler:

```
volume = v4l2_ctrl_new_std(&video_ctrl_handler, &ops, V4L2_CID_AUDIO_
↪VOLUME, ...);
v4l2_ctrl_new_std(&video_ctrl_handler, &ops, V4L2_CID_BRIGHTNESS, ...);
v4l2_ctrl_new_std(&video_ctrl_handler, &ops, V4L2_CID_CONTRAST, ...);
```

What you should not do is make two identical controls for two handlers. For example:

```
v4l2_ctrl_new_std(&radio_ctrl_handler, &radio_ops, V4L2_CID_AUDIO_MUTE, ...
↪);
v4l2_ctrl_new_std(&video_ctrl_handler, &video_ops, V4L2_CID_AUDIO_MUTE, ...
↪);
```

This would be bad since muting the radio would not change the video mute control. The rule is to have one control for each hardware ‘knob’ that you can twiddle.

Finding Controls

Normally you have created the controls yourself and you can store the struct v4l2_ctrl pointer into your own struct.

But sometimes you need to find a control from another handler that you do not own. For example, if you have to find a volume control from a subdev.

You can do that by calling v4l2_ctrl_find:

```
struct v4l2_ctrl *volume;

volume = v4l2_ctrl_find(sd->ctrl_handler, V4L2_CID_AUDIO_VOLUME);
```

Since v4l2_ctrl_find will lock the handler you have to be careful where you use it. For example, this is not a good idea:

```
struct v4l2_ctrl_handler ctrl_handler;

v4l2_ctrl_new_std(&ctrl_handler, &video_ops, V4L2_CID_BRIGHTNESS, ...);
v4l2_ctrl_new_std(&ctrl_handler, &video_ops, V4L2_CID_CONTRAST, ...);
```

···and in video_ops.s_ctrl:

```
case V4L2_CID_BRIGHTNESS:
        contrast = v4l2_find_ctrl(&ctrl_handler, V4L2_CID_CONTRAST);
        ...
```

When s_ctrl is called by the framework the ctrl_handler.lock is already taken, so attempting to find another control from the same handler will deadlock.

It is recommended not to use this function from inside the control ops.

Preventing Controls inheritance

When one control handler is added to another using v4l2_ctrl_add_handler, then by default all controls from one are merged to the other. But a subdev might have low-level controls that make sense for some advanced embedded system, but not when it is used in consumer-level hardware. In that case you want to keep those low-level controls local to the subdev. You can do this by simply setting the 'is_private' flag of the control to 1:

```
static const struct v4l2_ctrl_config ctrl_private = {
        .ops = &ctrl_custom_ops,
        .id = V4L2_CID_...,
        .name = "Some Private Control",
        .type = V4L2_CTRL_TYPE_INTEGER,
        .max = 15,
        .step = 1,
        .is_private = 1,
};

ctrl = v4l2_ctrl_new_custom(&foo->ctrl_handler, &ctrl_private, NULL);
```

These controls will now be skipped when v4l2_ctrl_add_handler is called.

V4L2_CTRL_TYPE_CTRL_CLASS Controls

Controls of this type can be used by GUIs to get the name of the control class. A fully featured GUI can make a dialog with multiple tabs with each tab containing the controls belonging to a particular control class. The name of each tab can be found by querying a special control with ID <control class | 1>.

Drivers do not have to care about this. The framework will automatically add a control of this type whenever the first control belonging to a new control class is added.

Adding Notify Callbacks

Sometimes the platform or bridge driver needs to be notified when a control from a sub-device driver changes. You can set a notify callback by calling this function:

```
void v4l2_ctrl_notify(struct v4l2_ctrl *ctrl,
        void (*notify)(struct v4l2_ctrl *ctrl, void *priv), void *priv);
```

Whenever the give control changes value the notify callback will be called with a pointer to the control and the priv pointer that was passed with v4l2_ctrl_notify. Note that the control's handler lock is held when the notify function is called.

There can be only one notify function per control handler. Any attempt to set another notify function will cause a WARN_ON.

v4l2_ctrl functions and data structures

union **v4l2_ctrl_ptr**
A pointer to a control value.

Definition

```
union v4l2_ctrl_ptr {
  s32 *p_s32;
  s64 *p_s64;
  u8 *p_u8;
  u16 *p_u16;
  u32 *p_u32;
  char *p_char;
  struct v4l2_ctrl_mpeg2_slice_params *p_mpeg2_slice_params;
  struct v4l2_ctrl_mpeg2_quantization *p_mpeg2_quantization;
  struct v4l2_ctrl_fwht_params *p_fwht_params;
  struct v4l2_ctrl_h264_sps *p_h264_sps;
  struct v4l2_ctrl_h264_pps *p_h264_pps;
  struct v4l2_ctrl_h264_scaling_matrix *p_h264_scaling_matrix;
  struct v4l2_ctrl_h264_slice_params *p_h264_slice_params;
  struct v4l2_ctrl_h264_decode_params *p_h264_decode_params;
  struct v4l2_ctrl_vp8_frame_header *p_vp8_frame_header;
  struct v4l2_ctrl_hevc_sps *p_hevc_sps;
  struct v4l2_ctrl_hevc_pps *p_hevc_pps;
  struct v4l2_ctrl_hevc_slice_params *p_hevc_slice_params;
  struct v4l2_area *p_area;
  void *p;
  const void *p_const;
};
```

Members

p_s32 Pointer to a 32-bit signed value.

p_s64 Pointer to a 64-bit signed value.

p_u8 Pointer to a 8-bit unsigned value.

p_u16 Pointer to a 16-bit unsigned value.

p_u32 Pointer to a 32-bit unsigned value.

p_char Pointer to a string.

p_mpeg2_slice_params Pointer to a MPEG2 slice parameters structure.

p_mpeg2_quantization Pointer to a MPEG2 quantization data structure.

p_fwht_params Pointer to a FWHT stateless parameters structure.

p_h264_sps Pointer to a struct v4l2_ctrl_h264_sps.

p_h264_pps Pointer to a struct v4l2_ctrl_h264_pps.

p_h264_scaling_matrix Pointer to a struct v4l2_ctrl_h264_scaling_matrix.

p_h264_slice_params Pointer to a struct v4l2_ctrl_h264_slice_params.

p_h264_decode_params Pointer to a struct v4l2_ctrl_h264_decode_params.

p_vp8_frame_header Pointer to a VP8 frame header structure.

p_hevc_sps Pointer to an HEVC sequence parameter set structure.

p_hevc_pps Pointer to an HEVC picture parameter set structure.

p_hevc_slice_params Pointer to an HEVC slice parameters structure.

p_area Pointer to an area.

p Pointer to a compound value.

p_const Pointer to a constant compound value.

union v4l2_ctrl_ptr **v4l2_ctrl_ptr_create**(void * ptr)
Helper function to return a v4l2_ctrl_ptr from a void pointer

Parameters

void * ptr The void pointer

struct **v4l2_ctrl_ops**
The control operations that the driver has to provide.

Definition

```
struct v4l2_ctrl_ops {
  int (*g_volatile_ctrl)(struct v4l2_ctrl *ctrl);
  int (*try_ctrl)(struct v4l2_ctrl *ctrl);
  int (*s_ctrl)(struct v4l2_ctrl *ctrl);
};
```

Members

g_volatile_ctrl Get a new value for this control. Generally only relevant for volatile (and usually read-only) controls such as a control that returns the current signal strength which changes continuously. If not set, then the currently cached value will be returned.

try_ctrl Test whether the control's value is valid. Only relevant when the usual min/max/step checks are not sufficient.

s_ctrl Actually set the new control value. s_ctrl is compulsory. The ctrl->handler->lock is held when these ops are called, so no one else can access controls owned by that handler.

struct **v4l2_ctrl_type_ops**
The control type operations that the driver has to provide.

Definition

```
struct v4l2_ctrl_type_ops {
  bool (*equal)(const struct v4l2_ctrl *ctrl, u32 idx,union v4l2_ctrl_ptr␣
↪ptr1, union v4l2_ctrl_ptr ptr2);
  void (*init)(const struct v4l2_ctrl *ctrl, u32 idx, union v4l2_ctrl_ptr␣
↪ptr);
  void (*log)(const struct v4l2_ctrl *ctrl);
  int (*validate)(const struct v4l2_ctrl *ctrl, u32 idx, union v4l2_ctrl_
↪ptr ptr);
};
```

Members

equal return true if both values are equal.

init initialize the value.

log log the value.

validate validate the value. Return 0 on success and a negative value otherwise.

v4l2_ctrl_notify_fnc
Typedef: typedef for a notify argument with a function that should be called when a control value has changed.

Syntax

```
void v4l2_ctrl_notify_fnc (struct v4l2_ctrl * ctrl, void *
priv);
```

Parameters

struct v4l2_ctrl * ctrl pointer to struct v4l2_ctrl

void * priv control private data

Description

This typedef definition is used as an argument to v4l2_ctrl_notify() and as an argument at struct v4l2_ctrl_handler.

struct **v4l2_ctrl**
The control structure.

Definition

```
struct v4l2_ctrl {
  struct list_head node;
  struct list_head ev_subs;
  struct v4l2_ctrl_handler *handler;
  struct v4l2_ctrl **cluster;
  unsigned int ncontrols;
  unsigned int done:1;
  unsigned int is_new:1;
  unsigned int has_changed:1;
  unsigned int is_private:1;
  unsigned int is_auto:1;
```

(continues on next page)

(continued from previous page)

```
  unsigned int is_int:1;
  unsigned int is_string:1;
  unsigned int is_ptr:1;
  unsigned int is_array:1;
  unsigned int has_volatiles:1;
  unsigned int call_notify:1;
  unsigned int manual_mode_value:8;
  const struct v4l2_ctrl_ops *ops;
  const struct v4l2_ctrl_type_ops *type_ops;
  u32 id;
  const char *name;
  enum v4l2_ctrl_type type;
  s64 minimum, maximum, default_value;
  u32 elems;
  u32 elem_size;
  u32 dims[V4L2_CTRL_MAX_DIMS];
  u32 nr_of_dims;
  union {
    u64 step;
    u64 menu_skip_mask;
  };
  union {
    const char * const *qmenu;
    const s64 *qmenu_int;
  };
  unsigned long flags;
  void *priv;
  s32 val;
  struct {
    s32 val;
  } cur;
  union v4l2_ctrl_ptr p_def;
  union v4l2_ctrl_ptr p_new;
  union v4l2_ctrl_ptr p_cur;
};
```

Members

node The list node.

ev_subs The list of control event subscriptions.

handler The handler that owns the control.

cluster Point to start of cluster array.

ncontrols Number of controls in cluster array.

done Internal flag: set for each processed control.

is_new Set when the user specified a new value for this control. It is also set when called from `v4l2_ctrl_handler_setup()`. Drivers should never set this flag.

has_changed Set when the current value differs from the new value. Drivers should never use this flag.

is_private If set, then this control is private to its handler and it will not be added to any other handlers. Drivers can set this flag.

is_auto If set, then this control selects whether the other cluster members are in ‘automatic’ mode or ‘manual’ mode. This is used for autogain/gain type clusters. Drivers should never set this flag directly.

is_int If set, then this control has a simple integer value (i.e. it uses ctrl->val).

is_string If set, then this control has type `V4L2_CTRL_TYPE_STRING`.

is_ptr If set, then this control is an array and/or has type >= `V4L2_CTRL_COMPOUND_TYPES` and/or has type `V4L2_CTRL_TYPE_STRING`. In other words, `struct v4l2_ext_control` uses field p to point to the data.

is_array If set, then this control contains an N-dimensional array.

has_volatiles If set, then one or more members of the cluster are volatile. Drivers should never touch this flag.

call_notify If set, then call the handler’s notify function whenever the control’s value changes.

manual_mode_value If the is_auto flag is set, then this is the value of the auto control that determines if that control is in manual mode. So if the value of the auto control equals this value, then the whole cluster is in manual mode. Drivers should never set this flag directly.

ops The control ops.

type_ops The control type ops.

id The control ID.

name The control name.

type The control type.

minimum The control’s minimum value.

maximum The control’s maximum value.

default_value The control’s default value.

elems The number of elements in the N-dimensional array.

elem_size The size in bytes of the control.

dims The size of each dimension.

nr_of_dims The number of dimensions in **dims**.

{unnamed_union} anonymous

step The control’s step value for non-menu controls.

menu_skip_mask The control’s skip mask for menu controls. This makes it easy to skip menu items that are not valid. If bit X is set, then menu item X is skipped. Of course, this only works for menus with <= 32 menu items. There are no menus that come close to that number, so this is OK. Should we ever need more, then this will have to be extended to a u64 or a bit array.

{unnamed_union} anonymous

qmenu A const char * array for all menu items. Array entries that are empty strings (“”) correspond to non-existing menu items (this is in addition to

the menu_skip_mask above). The last entry must be NULL. Used only if the **type** is V4L2_CTRL_TYPE_MENU.

qmenu_int A 64-bit integer array for with integer menu items. The size of array must be equal to the menu size, e. g.: $ceil(\frac{maximum-minimum}{step}) + 1$. Used only if the **type** is V4L2_CTRL_TYPE_INTEGER_MENU.

flags The control's flags.

priv The control's private pointer. For use by the driver. It is untouched by the control framework. Note that this pointer is not freed when the control is deleted. Should this be needed then a new internal bitfield can be added to tell the framework to free this pointer.

val The control's new s32 value.

cur Structure to store the current value.

cur.val The control's current value, if the **type** is represented via a u32 integer (see enum v4l2_ctrl_type).

p_def The control's default value represented via a union which provides a standard way of accessing control types through a pointer (for compound controls only).

p_new The control's new value represented via a union which provides a standard way of accessing control types through a pointer.

p_cur The control's current value represented via a union which provides a standard way of accessing control types through a pointer.

struct **v4l2_ctrl_ref**
The control reference.

Definition

```
struct v4l2_ctrl_ref {
  struct list_head node;
  struct v4l2_ctrl_ref *next;
  struct v4l2_ctrl *ctrl;
  struct v4l2_ctrl_helper *helper;
  bool from_other_dev;
  bool req_done;
  struct v4l2_ctrl_ref *req;
  union v4l2_ctrl_ptr p_req;
};
```

Members

node List node for the sorted list.

next Single-link list node for the hash.

ctrl The actual control information.

helper Pointer to helper struct. Used internally in prepare_ext_ctrls function at v4l2-ctrl.c.

from_other_dev If true, then **ctrl** was defined in another device than the struct v4l2_ctrl_handler.

req_done Internal flag: if the control handler containing this control reference is bound to a media request, then this is set when the control has been applied. This prevents applying controls from a cluster with multiple controls twice (when the first control of a cluster is applied, they all are).

req If set, this refers to another request that sets this control.

p_req If the control handler containing this control reference is bound to a media request, then this points to the value of the control that should be applied when the request is executed, or to the value of the control at the time that the request was completed.

Description

Each control handler has a list of these refs. The list_head is used to keep a sorted-by-control-ID list of all controls, while the next pointer is used to link the control in the hash's bucket.

struct **v4l2_ctrl_handler**
The control handler keeps track of all the controls: both the controls owned by the handler and those inherited from other handlers.

Definition

```
struct v4l2_ctrl_handler {
  struct mutex _lock;
  struct mutex *lock;
  struct list_head ctrls;
  struct list_head ctrl_refs;
  struct v4l2_ctrl_ref *cached;
  struct v4l2_ctrl_ref **buckets;
  v4l2_ctrl_notify_fnc notify;
  void *notify_priv;
  u16 nr_of_buckets;
  int error;
  bool request_is_queued;
  struct list_head requests;
  struct list_head requests_queued;
  struct media_request_object req_obj;
};
```

Members

_lock Default for "lock".

lock Lock to control access to this handler and its controls. May be replaced by the user right after init.

ctrls The list of controls owned by this handler.

ctrl_refs The list of control references.

cached The last found control reference. It is common that the same control is needed multiple times, so this is a simple optimization.

buckets Buckets for the hashing. Allows for quick control lookup.

notify A notify callback that is called whenever the control changes value. Note that the handler's lock is held when the notify function is called!

notify_priv Passed as argument to the v4l2_ctrl notify callback.

nr_of_buckets Total number of buckets in the array.

error The error code of the first failed control addition.

request_is_queued True if the request was queued.

requests List to keep track of open control handler request objects. For the parent control handler (**req_obj.req** == NULL) this is the list header. When the parent control handler is removed, it has to unbind and put all these requests since they refer to the parent.

requests_queued List of the queued requests. This determines the order in which these controls are applied. Once the request is completed it is removed from this list.

req_obj The `struct media_request_object`, used to link into a `struct media_request`. This request object has a refcount.

struct **v4l2_ctrl_config**
Control configuration structure.

Definition

```
struct v4l2_ctrl_config {
  const struct v4l2_ctrl_ops *ops;
  const struct v4l2_ctrl_type_ops *type_ops;
  u32 id;
  const char *name;
  enum v4l2_ctrl_type type;
  s64 min;
  s64 max;
  u64 step;
  s64 def;
  union v4l2_ctrl_ptr p_def;
  u32 dims[V4L2_CTRL_MAX_DIMS];
  u32 elem_size;
  u32 flags;
  u64 menu_skip_mask;
  const char * const *qmenu;
  const s64 *qmenu_int;
  unsigned int is_private:1;
};
```

Members

ops The control ops.

type_ops The control type ops. Only needed for compound controls.

id The control ID.

name The control name.

type The control type.

min The control’s minimum value.

max The control’s maximum value.

step The control’s step value for non-menu controls.

def The control’s default value.

p_def The control’s default value for compound controls.

dims The size of each dimension.

elem_size The size in bytes of the control.

flags The control’s flags.

menu_skip_mask The control’s skip mask for menu controls. This makes it easy to skip menu items that are not valid. If bit X is set, then menu item X is skipped. Of course, this only works for menus with <= 64 menu items. There are no menus that come close to that number, so this is OK. Should we ever need more, then this will have to be extended to a bit array.

qmenu A const char * array for all menu items. Array entries that are empty strings (“”) correspond to non-existing menu items (this is in addition to the menu_skip_mask above). The last entry must be NULL.

qmenu_int A const s64 integer array for all menu items of the type V4L2_CTRL_TYPE_INTEGER_MENU.

is_private If set, then this control is private to its handler and it will not be added to any other handlers.

void **v4l2_ctrl_fill**(u32 id, const char ** name, enum v4l2_ctrl_type * type, s64 * min, s64 * max, u64 * step, s64 * def, u32 * flags)

Fill in the control fields based on the control ID.

Parameters

u32 id ID of the control

const char ** name pointer to be filled with a string with the name of the control

enum v4l2_ctrl_type * type pointer for storing the type of the control

s64 * min pointer for storing the minimum value for the control

s64 * max pointer for storing the maximum value for the control

u64 * step pointer for storing the control step

s64 * def pointer for storing the default value for the control

u32 * flags pointer for storing the flags to be used on the control

Description

This works for all standard V4L2 controls. For non-standard controls it will only fill in the given arguments and **name** content will be set to `NULL`.

This function will overwrite the contents of **name**, **type** and **flags**. The contents of **min**, **max**, **step** and **def** may be modified depending on the type.

Note: Do not use in drivers! It is used internally for backwards compatibility control handling only. Once all drivers are converted to use the new control framework this function will no longer be exported.

---

int **v4l2_ctrl_handler_init_class**(struct v4l2_ctrl_handler * hdl, unsigned int nr_of_controls_hint, struct lock_class_key * key, const char * name)
Initialize the control handler.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

unsigned int nr_of_controls_hint A hint of how many controls this handler is expected to refer to. This is the total number, so including any inherited controls. It doesn't have to be precise, but if it is way off, then you either waste memory (too many buckets are allocated) or the control lookup becomes slower (not enough buckets are allocated, so there are more slow list lookups). It will always work, though.

struct lock_class_key * key Used by the lock validator if CONFIG_LOCKDEP is set.

const char * name Used by the lock validator if CONFIG_LOCKDEP is set.

Description

> **Attention:** Never use this call directly, always use the `v4l2_ctrl_handler_init()` macro that hides the **key** and **name** arguments.

Return

returns an error if the buckets could not be allocated. This error will also be stored in **hdl->error**.

v4l2_ctrl_handler_init(hdl, nr_of_controls_hint)
helper function to create a static struct `lock_class_key` and calls `v4l2_ctrl_handler_init_class()`

Parameters

hdl The control handler.

nr_of_controls_hint A hint of how many controls this handler is expected to refer to. This is the total number, so including any inherited controls. It doesn't have to be precise, but if it is way off, then you either waste memory (too many buckets are allocated) or the control lookup becomes slower (not enough buckets are allocated, so there are more slow list lookups). It will always work, though.

Description

This helper function creates a static struct `lock_class_key` and calls `v4l2_ctrl_handler_init_class()`, providing a proper name for the lock validador.

Use this helper function to initialize a control handler.

void **v4l2_ctrl_handler_free**(struct v4l2_ctrl_handler * hdl)
Free all controls owned by the handler and free the control list.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

Description

Does nothing if **hdl** == NULL.

void **v4l2_ctrl_lock**(struct v4l2_ctrl * ctrl)
Helper function to lock the handler associated with the control.

Parameters

struct v4l2_ctrl * ctrl The control to lock.

void **v4l2_ctrl_unlock**(struct v4l2_ctrl * ctrl)
Helper function to unlock the handler associated with the control.

Parameters

struct v4l2_ctrl * ctrl The control to unlock.

int **__v4l2_ctrl_handler_setup**(struct v4l2_ctrl_handler * hdl)
Call the s_ctrl op for all controls belonging to the handler to initialize the hardware to the current control values. The caller is responsible for acquiring the control handler mutex on behalf of `__v4l2_ctrl_handler_setup()`.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

Description

Button controls will be skipped, as are read-only controls.

If **hdl** == NULL, then this just returns 0.

int **v4l2_ctrl_handler_setup**(struct v4l2_ctrl_handler * hdl)
Call the s_ctrl op for all controls belonging to the handler to initialize the hardware to the current control values.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

Description

Button controls will be skipped, as are read-only controls.

If **hdl** == NULL, then this just returns 0.

void **v4l2_ctrl_handler_log_status**(struct v4l2_ctrl_handler * hdl, const char * prefix)
Log all controls owned by the handler.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

const char * prefix The prefix to use when logging the control values. If the prefix does not end with a space, then ": " will be added after the prefix. If **prefix** == NULL, then no prefix will be used.

Description

For use with VIDIOC_LOG_STATUS.

Does nothing if **hdl** == NULL.

struct v4l2_ctrl * **v4l2_ctrl_new_custom**(struct v4l2_ctrl_handler * hdl, const struct v4l2_ctrl_config * cfg, void * priv)

Allocate and initialize a new custom V4L2 control.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

const struct v4l2_ctrl_config * cfg The control's configuration data.

void * priv The control's driver-specific private data.

Description

If the `v4l2_ctrl` struct could not be allocated then NULL is returned and **hdl->error** is set to the error code (if it wasn't set already).

struct v4l2_ctrl * **v4l2_ctrl_new_std**(struct v4l2_ctrl_handler * hdl, const struct v4l2_ctrl_ops * ops, u32 id, s64 min, s64 max, u64 step, s64 def)

Allocate and initialize a new standard V4L2 non-menu control.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

const struct v4l2_ctrl_ops * ops The control ops.

u32 id The control ID.

s64 min The control's minimum value.

s64 max The control's maximum value.

u64 step The control's step value

s64 def The control's default value.

Description

If the `v4l2_ctrl` struct could not be allocated, or the control ID is not known, then NULL is returned and **hdl->error** is set to the appropriate error code (if it wasn't set already).

If **id** refers to a menu control, then this function will return NULL.

Use `v4l2_ctrl_new_std_menu()` when adding menu controls.

struct v4l2_ctrl * **v4l2_ctrl_new_std_menu**(struct v4l2_ctrl_handler * hdl, const struct v4l2_ctrl_ops * ops, u32 id, u8 max, u64 mask, u8 def)

Allocate and initialize a new standard V4L2 menu control.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

const struct v4l2_ctrl_ops * ops The control ops.

u32 id The control ID.

u8 max The control's maximum value.

u64 mask The control's skip mask for menu controls. This makes it easy to skip menu items that are not valid. If bit X is set, then menu item X is skipped. Of course, this only works for menus with <= 64 menu items. There are no menus that come close to that number, so this is OK. Should we ever need more, then this will have to be extended to a bit array.

u8 def The control's default value.

Description

Same as `v4l2_ctrl_new_std()`, but **min** is set to 0 and the **mask** value determines which menu items are to be skipped.

If **id** refers to a non-menu control, then this function will return NULL.

struct v4l2_ctrl * **v4l2_ctrl_new_std_menu_items**(struct v4l2_ctrl_handler * hdl, const struct v4l2_ctrl_ops * ops, u32 id, u8 max, u64 mask, u8 def, const char *const * qmenu)

Create a new standard V4L2 menu control with driver specific menu.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

const struct v4l2_ctrl_ops * ops The control ops.

u32 id The control ID.

u8 max The control's maximum value.

u64 mask The control's skip mask for menu controls. This makes it easy to skip menu items that are not valid. If bit X is set, then menu item X is skipped. Of course, this only works for menus with <= 64 menu items. There are no menus that come close to that number, so this is OK. Should we ever need more, then this will have to be extended to a bit array.

u8 def The control's default value.

const char *const * qmenu The new menu.

Description

Same as `v4l2_ctrl_new_std_menu()`, but **qmenu** will be the driver specific menu of this control.

struct v4l2_ctrl * **v4l2_ctrl_new_std_compound**(struct v4l2_ctrl_handler * hdl, const struct v4l2_ctrl_ops * ops, u32 id, const union v4l2_ctrl_ptr p_def)

Allocate and initialize a new standard V4L2 compound control.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

const struct v4l2_ctrl_ops * ops The control ops.

u32 id The control ID.

const union v4l2_ctrl_ptr p_def The control's default value.

Description

Sames as `v4l2_ctrl_new_std()`, but with support to compound controls, thanks to the **p_def** field. Use `v4l2_ctrl_ptr_create()` to create **p_def** from a pointer. Use v4l2_ctrl_ptr_create(NULL) if the default value of the compound control should be all zeroes.

struct v4l2_ctrl * **v4l2_ctrl_new_int_menu**(struct v4l2_ctrl_handler * hdl, const struct v4l2_ctrl_ops * ops, u32 id, u8 max, u8 def, const s64 * qmenu_int)

Create a new standard V4L2 integer menu control.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

const struct v4l2_ctrl_ops * ops The control ops.

u32 id The control ID.

u8 max The control's maximum value.

u8 def The control's default value.

const s64 * qmenu_int The control's menu entries.

Description

Same as `v4l2_ctrl_new_std_menu()`, but **mask** is set to 0 and it additionally takes as an argument an array of integers determining the menu items.

If **id** refers to a non-integer-menu control, then this function will return `NULL`.

v4l2_ctrl_filter

Typedef: Typedef to define the filter function to be used when adding a control handler.

Syntax

```
bool v4l2_ctrl_filter (const struct v4l2_ctrl * ctrl);
```

Parameters

const struct v4l2_ctrl * ctrl pointer to struct `v4l2_ctrl`.

int **v4l2_ctrl_add_handler**(struct v4l2_ctrl_handler * hdl, struct v4l2_ctrl_handler * add, v4l2_ctrl_filter filter, bool from_other_dev)

Add all controls from handler **add** to handler **hdl**.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

struct v4l2_ctrl_handler * add The control handler whose controls you want to add to the **hdl** control handler.

v4l2_ctrl_filter filter This function will filter which controls should be added.

bool from_other_dev If true, then the controls in **add** were defined in another device than **hdl**.

Description

Does nothing if either of the two handlers is a NULL pointer. If **filter** is NULL, then all controls are added. Otherwise only those controls for which **filter** returns true will be added. In case of an error **hdl->error** will be set to the error code (if it wasn't set already).

bool **v4l2_ctrl_radio_filter**(const struct v4l2_ctrl * ctrl)
Standard filter for radio controls.

Parameters

const struct v4l2_ctrl * ctrl The control that is filtered.

Description

This will return true for any controls that are valid for radio device nodes. Those are all of the V4L2_CID_AUDIO_* user controls and all FM transmitter class controls.

This function is to be used with `v4l2_ctrl_add_handler()`.

void **v4l2_ctrl_cluster**(unsigned int ncontrols, struct v4l2_ctrl ** controls)
Mark all controls in the cluster as belonging to that cluster.

Parameters

unsigned int ncontrols The number of controls in this cluster.

struct v4l2_ctrl ** controls The cluster control array of size **ncontrols**.

void **v4l2_ctrl_auto_cluster**(unsigned int ncontrols, struct v4l2_ctrl ** controls, u8 manual_val, bool set_volatile)
Mark all controls in the cluster as belonging to that cluster and set it up for autofoo/foo-type handling.

Parameters

unsigned int ncontrols The number of controls in this cluster.

struct v4l2_ctrl ** controls The cluster control array of size **ncontrols**. The first control must be the 'auto' control (e.g. autogain, autoexposure, etc.)

u8 manual_val The value for the first control in the cluster that equals the manual setting.

bool set_volatile If true, then all controls except the first auto control will be volatile.

Description

Use for control groups where one control selects some automatic feature and the other controls are only active whenever the automatic feature is turned off (manual mode). Typical examples: autogain vs gain, auto-whitebalance vs red and blue balance, etc.

The behavior of such controls is as follows:

When the autofoo control is set to automatic, then any manual controls are set to inactive and any reads will call g_volatile_ctrl (if the control was marked volatile).

When the autofoo control is set to manual, then any manual controls will be marked active, and any reads will just return the current value without going through g_volatile_ctrl.

In addition, this function will set the `V4L2_CTRL_FLAG_UPDATE` flag on the autofoo control and `V4L2_CTRL_FLAG_INACTIVE` on the foo control(s) if autofoo is in auto mode.

struct v4l2_ctrl * **v4l2_ctrl_find**(struct v4l2_ctrl_handler * hdl, u32 id)
Find a control with the given ID.

Parameters

struct v4l2_ctrl_handler * hdl The control handler.

u32 id The control ID to find.

Description

If **hdl** == NULL this will return NULL as well. Will lock the handler so do not use from inside `v4l2_ctrl_ops`.

void **v4l2_ctrl_activate**(struct v4l2_ctrl * ctrl, bool active)
Make the control active or inactive.

Parameters

struct v4l2_ctrl * ctrl The control to (de)activate.

bool active True if the control should become active.

Description

This sets or clears the V4L2_CTRL_FLAG_INACTIVE flag atomically. Does nothing if **ctrl** == NULL. This will usually be called from within the s_ctrl op. The V4L2_EVENT_CTRL event will be generated afterwards.

This function assumes that the control handler is locked.

void **__v4l2_ctrl_grab**(struct v4l2_ctrl * ctrl, bool grabbed)
Unlocked variant of v4l2_ctrl_grab.

Parameters

struct v4l2_ctrl * ctrl The control to (de)activate.

bool grabbed True if the control should become grabbed.

Description

This sets or clears the V4L2_CTRL_FLAG_GRABBED flag atomically. Does nothing if **ctrl** == NULL. The V4L2_EVENT_CTRL event will be generated afterwards. This will usually be called when starting or stopping streaming in the driver.

This function assumes that the control handler is locked by the caller.

void **v4l2_ctrl_grab**(struct v4l2_ctrl * ctrl, bool grabbed)
Mark the control as grabbed or not grabbed.

Parameters

struct v4l2_ctrl * ctrl The control to (de)activate.

bool grabbed True if the control should become grabbed.

Description

This sets or clears the V4L2_CTRL_FLAG_GRABBED flag atomically. Does nothing if **ctrl** == NULL. The V4L2_EVENT_CTRL event will be generated afterwards. This will usually be called when starting or stopping streaming in the driver.

This function assumes that the control handler is not locked and will take the lock itself.

int **__v4l2_ctrl_modify_range**(struct v4l2_ctrl * ctrl, s64 min, s64 max, u64 step, s64 def)
Unlocked variant of `v4l2_ctrl_modify_range()`

Parameters

struct v4l2_ctrl * ctrl The control to update.

s64 min The control's minimum value.

s64 max The control's maximum value.

u64 step The control's step value

s64 def The control's default value.

Description

Update the range of a control on the fly. This works for control types INTEGER, BOOLEAN, MENU, INTEGER MENU and BITMASK. For menu controls the **step** value is interpreted as a menu_skip_mask.

An error is returned if one of the range arguments is invalid for this control type.

The caller is responsible for acquiring the control handler mutex on behalf of `__v4l2_ctrl_modify_range()`.

int **v4l2_ctrl_modify_range**(struct v4l2_ctrl * ctrl, s64 min, s64 max, u64 step, s64 def)
Update the range of a control.

Parameters

struct v4l2_ctrl * ctrl The control to update.

s64 min The control's minimum value.

s64 max The control's maximum value.

u64 step The control's step value

s64 def The control's default value.

Description

Update the range of a control on the fly. This works for control types INTEGER, BOOLEAN, MENU, INTEGER MENU and BITMASK. For menu controls the **step** value is interpreted as a menu_skip_mask.

An error is returned if one of the range arguments is invalid for this control type.

This function assumes that the control handler is not locked and will take the lock itself.

void **v4l2_ctrl_notify**(struct v4l2_ctrl * ctrl, v4l2_ctrl_notify_fnc notify, void * priv)
Function to set a notify callback for a control.

Parameters

struct v4l2_ctrl * ctrl The control.

v4l2_ctrl_notify_fnc notify The callback function.

void * priv The callback private handle, passed as argument to the callback.

Description

This function sets a callback function for the control. If **ctrl** is NULL, then it will do nothing. If **notify** is NULL, then the notify callback will be removed.

There can be only one notify. If another already exists, then a WARN_ON will be issued and the function will do nothing.

const char * **v4l2_ctrl_get_name**(u32 id)
Get the name of the control

Parameters

u32 id The control ID.

Description

This function returns the name of the given control ID or NULL if it isn't a known control.

const char * const * **v4l2_ctrl_get_menu**(u32 id)
Get the menu string array of the control

Parameters

u32 id The control ID.

Description

This function returns the NULL-terminated menu string array name of the given control ID or NULL if it isn't a known menu control.

const s64 * **v4l2_ctrl_get_int_menu**(u32 id, u32 * len)
Get the integer menu array of the control

Parameters

u32 id The control ID.

u32 * len The size of the integer array.

Description

This function returns the integer array of the given control ID or NULL if it if it isn't a known integer menu control.

s32 **v4l2_ctrl_g_ctrl**(struct v4l2_ctrl * ctrl)
Helper function to get the control's value from within a driver.

Parameters

struct v4l2_ctrl * ctrl The control.

Description

This returns the control's value safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for integer type controls only.

int **__v4l2_ctrl_s_ctrl**(struct v4l2_ctrl * ctrl, s32 val)
Unlocked variant of `v4l2_ctrl_s_ctrl()`.

Parameters

struct v4l2_ctrl * ctrl The control.

s32 val The new value.

Description

This sets the control's new value safely by going through the control framework. This function assumes the control's handler is already locked, allowing it to be used from within the `v4l2_ctrl_ops` functions.

This function is for integer type controls only.

int **v4l2_ctrl_s_ctrl**(struct v4l2_ctrl * ctrl, s32 val)
Helper function to set the control's value from within a driver.

Parameters

struct v4l2_ctrl * ctrl The control.

s32 val The new value.

Description

This sets the control's new value safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for integer type controls only.

s64 **v4l2_ctrl_g_ctrl_int64**(struct v4l2_ctrl * ctrl)
Helper function to get a 64-bit control's value from within a driver.

Parameters

struct v4l2_ctrl * ctrl The control.

Description

This returns the control's value safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for 64-bit integer type controls only.

int **__v4l2_ctrl_s_ctrl_int64**(struct v4l2_ctrl * ctrl, s64 val)
Unlocked variant of `v4l2_ctrl_s_ctrl_int64()`.

Parameters

struct v4l2_ctrl * ctrl The control.

s64 val The new value.

Description

This sets the control's new value safely by going through the control framework. This function assumes the control's handler is already locked, allowing it to be used from within the `v4l2_ctrl_ops` functions.

This function is for 64-bit integer type controls only.

int **v4l2_ctrl_s_ctrl_int64**(struct v4l2_ctrl * ctrl, s64 val)
Helper function to set a 64-bit control's value from within a driver.

Parameters

struct v4l2_ctrl * ctrl The control.

s64 val The new value.

Description

This sets the control's new value safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for 64-bit integer type controls only.

int **__v4l2_ctrl_s_ctrl_string**(struct v4l2_ctrl * ctrl, const char * s)
Unlocked variant of `v4l2_ctrl_s_ctrl_string()`.

Parameters

struct v4l2_ctrl * ctrl The control.

const char * s The new string.

Description

This sets the control's new string safely by going through the control framework. This function assumes the control's handler is already locked, allowing it to be used from within the `v4l2_ctrl_ops` functions.

This function is for string type controls only.

int **v4l2_ctrl_s_ctrl_string**(struct v4l2_ctrl * ctrl, const char * s)
Helper function to set a control's string value from within a driver.

Parameters

struct v4l2_ctrl * ctrl The control.

const char * s The new string.

Description

This sets the control's new string safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for string type controls only.

int **__v4l2_ctrl_s_ctrl_compound**(struct v4l2_ctrl * ctrl, enum v4l2_ctrl_type type, const void * p)

Unlocked variant to set a compound control

Parameters

struct v4l2_ctrl * ctrl The control.

enum v4l2_ctrl_type type The type of the data.

const void * p The new compound payload.

Description

This sets the control's new compound payload safely by going through the control framework. This function assumes the control's handler is already locked, allowing it to be used from within the `v4l2_ctrl_ops` functions.

This function is for compound type controls only.

int **v4l2_ctrl_s_ctrl_compound**(struct v4l2_ctrl * ctrl, enum v4l2_ctrl_type type, const void * p)

Helper function to set a compound control from within a driver.

Parameters

struct v4l2_ctrl * ctrl The control.

enum v4l2_ctrl_type type The type of the data.

const void * p The new compound payload.

Description

This sets the control's new compound payload safely by going through the control framework. This function will lock the control's handler, so it cannot be used from within the `v4l2_ctrl_ops` functions.

This function is for compound type controls only.

void **v4l2_ctrl_replace**(struct v4l2_event * old, const struct v4l2_event * new)

Function to be used as a callback to `struct v4l2_subscribed_event_ops` replace()

Parameters

struct v4l2_event * old pointer to struct `v4l2_event` with the reported event;

const struct v4l2_event * new pointer to struct `v4l2_event` with the modified event;

void **v4l2_ctrl_merge**(const struct v4l2_event * old, struct v4l2_event * new)
Function to be used as a callback to `struct v4l2_subscribed_event_ops` merge()

Parameters

const struct v4l2_event * old pointer to struct `v4l2_event` with the reported event;

struct v4l2_event * new pointer to struct `v4l2_event` with the merged event;

int **v4l2_ctrl_log_status**(struct file * file, void * fh)
helper function to implement `VIDIOC_LOG_STATUS` ioctl

Parameters

struct file * file pointer to struct file

void * fh unused. Kept just to be compatible to the arguments expected by `struct v4l2_ioctl_ops`.vidioc_log_status.

Description

Can be used as a vidioc_log_status function that just dumps all controls associated with the filehandle.

int **v4l2_ctrl_subscribe_event**(struct v4l2_fh * fh, const struct v4l2_event_subscription * sub)
Subscribes to an event

Parameters

struct v4l2_fh * fh pointer to struct v4l2_fh

const struct v4l2_event_subscription * sub pointer to `struct v4l2_event_subscription`

Description

Can be used as a vidioc_subscribe_event function that just subscribes control events.

__poll_t **v4l2_ctrl_poll**(struct file * file, struct poll_table_struct * wait)
function to be used as a callback to the poll() That just polls for control events.

Parameters

struct file * file pointer to struct file

struct poll_table_struct * wait pointer to struct poll_table_struct

int **v4l2_ctrl_request_setup**(struct media_request * req, struct v4l2_ctrl_handler * parent)
helper function to apply control values in a request

Parameters

struct media_request * req The request

struct v4l2_ctrl_handler * parent The parent control handler (‘priv’ in `media_request_object_find()`)

Description

This is a helper function to call the control handler' s s_ctrl callback with the control values contained in the request. Do note that this approach of applying control values in a request is only applicable to memory-to-memory devices.

void **v4l2_ctrl_request_complete**(struct media_request * req, struct v4l2_ctrl_handler * parent)
Complete a control handler request object

Parameters

struct media_request * req The request

struct v4l2_ctrl_handler * parent The parent control handler ('priv' in `media_request_object_find()`)

Description

This function is to be called on each control handler that may have had a request object associated with it, i.e. control handlers of a driver that supports requests.

The function first obtains the values of any volatile controls in the control handler and attach them to the request. Then, the function completes the request object.

struct v4l2_ctrl_handler * **v4l2_ctrl_request_hdl_find**(struct media_request * req, struct v4l2_ctrl_handler * parent)
Find the control handler in the request

Parameters

struct media_request * req The request

struct v4l2_ctrl_handler * parent The parent control handler ('priv' in `media_request_object_find()`)

Description

This function finds the control handler in the request. It may return NULL if not found. When done, you must call v4l2_ctrl_request_put_hdl() with the returned handler pointer.

If the request is not in state VALIDATING or QUEUED, then this function will always return NULL.

Note that in state VALIDATING the req_queue_mutex is held, so no objects can be added or deleted from the request.

In state QUEUED it is the driver that will have to ensure this.

void **v4l2_ctrl_request_hdl_put**(struct v4l2_ctrl_handler * hdl)
Put the control handler

Parameters

struct v4l2_ctrl_handler * hdl Put this control handler

Description

This function released the control handler previously obtained from' `v4l2_ctrl_request_hdl_find()`.

struct v4l2_ctrl * **v4l2_ctrl_request_hdl_ctrl_find**(struct v4l2_ctrl_handler * hdl, u32 id)

Find a control with the given ID.

Parameters

`struct v4l2_ctrl_handler * hdl` The control handler from the request.

`u32 id` The ID of the control to find.

Description

This function returns a pointer to the control if this control is part of the request or NULL otherwise.

int **v4l2_queryctrl**(struct v4l2_ctrl_handler * hdl, struct v4l2_queryctrl * qc)

Helper function to implement VIDIOC_QUERYCTRL ioctl

Parameters

`struct v4l2_ctrl_handler * hdl` pointer to `struct v4l2_ctrl_handler`

`struct v4l2_queryctrl * qc` pointer to `struct v4l2_queryctrl`

Description

If hdl == NULL then they will all return -EINVAL.

int **v4l2_query_ext_ctrl**(struct v4l2_ctrl_handler * hdl, struct v4l2_query_ext_ctrl * qc)

Helper function to implement VIDIOC_QUERY_EXT_CTRL ioctl

Parameters

`struct v4l2_ctrl_handler * hdl` pointer to `struct v4l2_ctrl_handler`

`struct v4l2_query_ext_ctrl * qc` pointer to `struct v4l2_query_ext_ctrl`

Description

If hdl == NULL then they will all return -EINVAL.

int **v4l2_querymenu**(struct v4l2_ctrl_handler * hdl, struct v4l2_querymenu * qm)

Helper function to implement VIDIOC_QUERYMENU ioctl

Parameters

`struct v4l2_ctrl_handler * hdl` pointer to `struct v4l2_ctrl_handler`

`struct v4l2_querymenu * qm` pointer to `struct v4l2_querymenu`

Description

If hdl == NULL then they will all return -EINVAL.

int **v4l2_g_ctrl**(struct v4l2_ctrl_handler * hdl, struct v4l2_control * ctrl)

Helper function to implement VIDIOC_G_CTRL ioctl

Parameters

struct v4l2_ctrl_handler * hdl pointer to struct v4l2_ctrl_handler

struct v4l2_control * ctrl pointer to struct v4l2_control

Description

If hdl == NULL then they will all return -EINVAL.

int **v4l2_s_ctrl**(struct v4l2_fh * fh, struct v4l2_ctrl_handler * hdl, struct v4l2_control * ctrl)

Helper function to implement VIDIOC_S_CTRL ioctl

Parameters

struct v4l2_fh * fh pointer to struct v4l2_fh

struct v4l2_ctrl_handler * hdl pointer to struct v4l2_ctrl_handler

struct v4l2_control * ctrl pointer to struct v4l2_control

Description

If hdl == NULL then they will all return -EINVAL.

int **v4l2_g_ext_ctrls**(struct v4l2_ctrl_handler * hdl, struct video_device * vdev, struct media_device * mdev, struct v4l2_ext_controls * c)

Helper function to implement VIDIOC_G_EXT_CTRLS ioctl

Parameters

struct v4l2_ctrl_handler * hdl pointer to struct v4l2_ctrl_handler

struct video_device * vdev pointer to struct video_device

struct media_device * mdev pointer to struct media_device

struct v4l2_ext_controls * c pointer to struct v4l2_ext_controls

Description

If hdl == NULL then they will all return -EINVAL.

int **v4l2_try_ext_ctrls**(struct v4l2_ctrl_handler * hdl, struct video_device * vdev, struct media_device * mdev, struct v4l2_ext_controls * c)

Helper function to implement VIDIOC_TRY_EXT_CTRLS ioctl

Parameters

struct v4l2_ctrl_handler * hdl pointer to struct v4l2_ctrl_handler

struct video_device * vdev pointer to struct video_device

struct media_device * mdev pointer to struct media_device

struct v4l2_ext_controls * c pointer to struct v4l2_ext_controls

Description

If hdl == NULL then they will all return -EINVAL.

int **v4l2_s_ext_ctrls**(struct v4l2_fh * fh, struct v4l2_ctrl_handler * hdl, struct video_device * vdev, struct media_device * mdev, struct v4l2_ext_controls * c)
Helper function to implement VIDIOC_S_EXT_CTRLS ioctl

Parameters

struct v4l2_fh * fh pointer to `struct v4l2_fh`

struct v4l2_ctrl_handler * hdl pointer to `struct v4l2_ctrl_handler`

struct video_device * vdev pointer to `struct video_device`

struct media_device * mdev pointer to `struct media_device`

struct v4l2_ext_controls * c pointer to `struct v4l2_ext_controls`

Description

If hdl == NULL then they will all return -EINVAL.

int **v4l2_ctrl_subdev_subscribe_event**(struct v4l2_subdev * sd, struct v4l2_fh * fh, struct v4l2_event_subscription * sub)
Helper function to implement as a `struct v4l2_subdev_core_ops` subscribe_event function that just subscribes control events.

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

struct v4l2_fh * fh pointer to `struct v4l2_fh`

struct v4l2_event_subscription * sub pointer to `struct v4l2_event_subscription`

int **v4l2_ctrl_subdev_log_status**(struct v4l2_subdev * sd)
Log all controls owned by subdev's control handler.

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

int **v4l2_ctrl_new_fwnode_properties**(struct v4l2_ctrl_handler * hdl, const struct v4l2_ctrl_ops * ctrl_ops, const struct v4l2_fwnode_device_properties * p)
Register controls for the device properties

Parameters

struct v4l2_ctrl_handler * hdl pointer to `struct v4l2_ctrl_handler` to register controls on

const struct v4l2_ctrl_ops * ctrl_ops pointer to `struct v4l2_ctrl_ops` to register controls with

const struct v4l2_fwnode_device_properties * p pointer to `struct v4l2_fwnode_device_properties`

Description

This function registers controls associated to device properties, using the property values contained in **p** parameter, if the property has been set to a value.

Currently the following v4l2 controls are parsed and registered: - V4L2_CID_CAMERA_ORIENTATION - V4L2_CID_CAMERA_SENSOR_ROTATION;

Controls already registered by the caller with the **hdl** control handler are not overwritten. Callers should register the controls they want to handle themselves before calling this function.

Return

0 on success, a negative error code on failure.

53.1.14 Videobuf Framework

Author: Jonathan Corbet <corbet@lwn.net>

Current as of 2.6.33

Note: The videobuf framework was deprecated in favor of videobuf2. Shouldn't be used on new drivers.

Introduction

The videobuf layer functions as a sort of glue layer between a V4L2 driver and user space. It handles the allocation and management of buffers for the storage of video frames. There is a set of functions which can be used to implement many of the standard POSIX I/O system calls, including read(), poll(), and, happily, mmap(). Another set of functions can be used to implement the bulk of the V4L2 ioctl() calls related to streaming I/O, including buffer allocation, queueing and dequeueing, and streaming control. Using videobuf imposes a few design decisions on the driver author, but the payback comes in the form of reduced code in the driver and a consistent implementation of the V4L2 user-space API.

Buffer types

Not all video devices use the same kind of buffers. In fact, there are (at least) three common variations:

- Buffers which are scattered in both the physical and (kernel) virtual address spaces. (Almost) all user-space buffers are like this, but it makes great sense to allocate kernel-space buffers this way as well when it is possible. Unfortunately, it is not always possible; working with this kind of buffer normally requires hardware which can do scatter/gather DMA operations.
- Buffers which are physically scattered, but which are virtually contiguous; buffers allocated with vmalloc(), in other words. These buffers are just as hard to use for DMA operations, but they can be useful in situations where DMA is not available but virtually-contiguous buffers are convenient.

- Buffers which are physically contiguous. Allocation of this kind of buffer can be unreliable on fragmented systems, but simpler DMA controllers cannot deal with anything else.

Videobuf can work with all three types of buffers, but the driver author must pick one at the outset and design the driver around that decision.

[It's worth noting that there's a fourth kind of buffer: "overlay" buffers which are located within the system's video memory. The overlay functionality is considered to be deprecated for most use, but it still shows up occasionally in system-on-chip drivers where the performance benefits merit the use of this technique. Overlay buffers can be handled as a form of scattered buffer, but there are very few implementations in the kernel and a description of this technique is currently beyond the scope of this document.]

Data structures, callbacks, and initialization

Depending on which type of buffers are being used, the driver should include one of the following files:

```
<media/videobuf-dma-sg.h>           /* Physically scattered */
<media/videobuf-vmalloc.h>          /* vmalloc() buffers    */
<media/videobuf-dma-contig.h>       /* Physically contiguous */
```

The driver's data structure describing a V4L2 device should include a struct videobuf_queue instance for the management of the buffer queue, along with a list_head for the queue of available buffers. There will also need to be an interrupt-safe spinlock which is used to protect (at least) the queue.

The next step is to write four simple callbacks to help videobuf deal with the management of buffers:

```
struct videobuf_queue_ops {
    int (*buf_setup)(struct videobuf_queue *q,
                     unsigned int *count, unsigned int *size);
    int (*buf_prepare)(struct videobuf_queue *q,
                       struct videobuf_buffer *vb,
                       enum v4l2_field field);
    void (*buf_queue)(struct videobuf_queue *q,
                      struct videobuf_buffer *vb);
    void (*buf_release)(struct videobuf_queue *q,
                        struct videobuf_buffer *vb);
};
```

buf_setup() is called early in the I/O process, when streaming is being initiated; its purpose is to tell videobuf about the I/O stream. The count parameter will be a suggested number of buffers to use; the driver should check it for rationality and adjust it if need be. As a practical rule, a minimum of two buffers are needed for proper streaming, and there is usually a maximum (which cannot exceed 32) which makes sense for each device. The size parameter should be set to the expected (maximum) size for each frame of data.

Each buffer (in the form of a struct videobuf_buffer pointer) will be passed to buf_prepare(), which should set the buffer's size, width, height, and field fields

properly. If the buffer's state field is VIDEOBUF_NEEDS_INIT, the driver should pass it to:

```
int videobuf_iolock(struct videobuf_queue* q, struct videobuf_buffer *vb,
                    struct v4l2_framebuffer *fbuf);
```

Among other things, this call will usually allocate memory for the buffer. Finally, the buf_prepare() function should set the buffer's state to VIDEOBUF_PREPARED.

When a buffer is queued for I/O, it is passed to buf_queue(), which should put it onto the driver's list of available buffers and set its state to VIDEOBUF_QUEUED. Note that this function is called with the queue spinlock held; if it tries to acquire it as well things will come to a screeching halt. Yes, this is the voice of experience. Note also that videobuf may wait on the first buffer in the queue; placing other buffers in front of it could again gum up the works. So use list_add_tail() to enqueue buffers.

Finally, buf_release() is called when a buffer is no longer intended to be used. The driver should ensure that there is no I/O active on the buffer, then pass it to the appropriate free routine(s):

```
/* Scatter/gather drivers */
int videobuf_dma_unmap(struct videobuf_queue *q,
                       struct videobuf_dmabuf *dma);
int videobuf_dma_free(struct videobuf_dmabuf *dma);

/* vmalloc drivers */
void videobuf_vmalloc_free (struct videobuf_buffer *buf);

/* Contiguous drivers */
void videobuf_dma_contig_free(struct videobuf_queue *q,
                              struct videobuf_buffer *buf);
```

One way to ensure that a buffer is no longer under I/O is to pass it to:

```
int videobuf_waiton(struct videobuf_buffer *vb, int non_blocking, int␣
↪intr);
```

Here, vb is the buffer, non_blocking indicates whether non-blocking I/O should be used (it should be zero in the buf_release() case), and intr controls whether an interruptible wait is used.

File operations

At this point, much of the work is done; much of the rest is slipping videobuf calls into the implementation of the other driver callbacks. The first step is in the open() function, which must initialize the videobuf queue. The function to use depends on the type of buffer used:

```
void videobuf_queue_sg_init(struct videobuf_queue *q,
                            struct videobuf_queue_ops *ops,
                            struct device *dev,
                            spinlock_t *irqlock,
                            enum v4l2_buf_type type,
                            enum v4l2_field field,
```

(continues on next page)

(continued from previous page)

```
                              unsigned int msize,
                              void *priv);

void videobuf_queue_vmalloc_init(struct videobuf_queue *q,
                              struct videobuf_queue_ops *ops,
                              struct device *dev,
                              spinlock_t *irqlock,
                              enum v4l2_buf_type type,
                              enum v4l2_field field,
                              unsigned int msize,
                              void *priv);

void videobuf_queue_dma_contig_init(struct videobuf_queue *q,
                                    struct videobuf_queue_ops *ops,
                                    struct device *dev,
                                    spinlock_t *irqlock,
                                    enum v4l2_buf_type type,
                                    enum v4l2_field field,
                                    unsigned int msize,
                                    void *priv);
```

In each case, the parameters are the same: q is the queue structure for the device, ops is the set of callbacks as described above, dev is the device structure for this video device, irqlock is an interrupt-safe spinlock to protect access to the data structures, type is the buffer type used by the device (cameras will use V4L2_BUF_TYPE_VIDEO_CAPTURE, for example), field describes which field is being captured (often V4L2_FIELD_NONE for progressive devices), msize is the size of any containing structure used around struct videobuf_buffer, and priv is a private data pointer which shows up in the priv_data field of struct videobuf_queue. Note that these are void functions which, evidently, are immune to failure.

V4L2 capture drivers can be written to support either of two APIs: the read() system call and the rather more complicated streaming mechanism. As a general rule, it is necessary to support both to ensure that all applications have a chance of working with the device. Videobuf makes it easy to do that with the same code. To implement read(), the driver need only make a call to one of:

```
ssize_t videobuf_read_one(struct videobuf_queue *q,
                          char __user *data, size_t count,
                          loff_t *ppos, int nonblocking);

ssize_t videobuf_read_stream(struct videobuf_queue *q,
                             char __user *data, size_t count,
                             loff_t *ppos, int vbihack, int nonblocking);
```

Either one of these functions will read frame data into data, returning the amount actually read; the difference is that videobuf_read_one() will only read a single frame, while videobuf_read_stream() will read multiple frames if they are needed to satisfy the count requested by the application. A typical driver read() implementation will start the capture engine, call one of the above functions, then stop the engine before returning (though a smarter implementation might leave the engine running for a little while in anticipation of another read() call happening in the near future).

The poll() function can usually be implemented with a direct call to:

```
unsigned int videobuf_poll_stream(struct file *file,
                                  struct videobuf_queue *q,
                                  poll_table *wait);
```

Note that the actual wait queue eventually used will be the one associated with the first available buffer.

When streaming I/O is done to kernel-space buffers, the driver must support the mmap() system call to enable user space to access the data. In many V4L2 drivers, the often-complex mmap() implementation simplifies to a single call to:

```
int videobuf_mmap_mapper(struct videobuf_queue *q,
                         struct vm_area_struct *vma);
```

Everything else is handled by the videobuf code.

The release() function requires two separate videobuf calls:

```
void videobuf_stop(struct videobuf_queue *q);
int videobuf_mmap_free(struct videobuf_queue *q);
```

The call to videobuf_stop() terminates any I/O in progress - though it is still up to the driver to stop the capture engine. The call to videobuf_mmap_free() will ensure that all buffers have been unmapped; if so, they will all be passed to the buf_release() callback. If buffers remain mapped, videobuf_mmap_free() returns an error code instead. The purpose is clearly to cause the closing of the file descriptor to fail if buffers are still mapped, but every driver in the 2.6.32 kernel cheerfully ignores its return value.

ioctl() operations

The V4L2 API includes a very long list of driver callbacks to respond to the many ioctl() commands made available to user space. A number of these - those associated with streaming I/O - turn almost directly into videobuf calls. The relevant helper functions are:

```
int videobuf_reqbufs(struct videobuf_queue *q,
                     struct v4l2_requestbuffers *req);
int videobuf_querybuf(struct videobuf_queue *q, struct v4l2_buffer *b);
int videobuf_qbuf(struct videobuf_queue *q, struct v4l2_buffer *b);
int videobuf_dqbuf(struct videobuf_queue *q, struct v4l2_buffer *b,
                   int nonblocking);
int videobuf_streamon(struct videobuf_queue *q);
int videobuf_streamoff(struct videobuf_queue *q);
```

So, for example, a VIDIOC_REQBUFS call turns into a call to the driver's vidioc_reqbufs() callback which, in turn, usually only needs to locate the proper struct videobuf_queue pointer and pass it to videobuf_reqbufs(). These support functions can replace a great deal of buffer management boilerplate in a lot of V4L2 drivers.

The vidioc_streamon() and vidioc_streamoff() functions will be a bit more complex, of course, since they will also need to deal with starting and stopping the capture engine.

Buffer allocation

Thus far, we have talked about buffers, but have not looked at how they are allocated. The scatter/gather case is the most complex on this front. For allocation, the driver can leave buffer allocation entirely up to the videobuf layer; in this case, buffers will be allocated as anonymous user-space pages and will be very scattered indeed. If the application is using user-space buffers, no allocation is needed; the videobuf layer will take care of calling get_user_pages() and filling in the scatterlist array.

If the driver needs to do its own memory allocation, it should be done in the vidioc_reqbufs() function, after calling videobuf_reqbufs(). The first step is a call to:

```
struct videobuf_dmabuf *videobuf_to_dma(struct videobuf_buffer *buf);
```

The returned videobuf_dmabuf structure (defined in <media/videobuf-dma-sg.h>) includes a couple of relevant fields:

```
struct scatterlist  *sglist;
int                 sglen;
```

The driver must allocate an appropriately-sized scatterlist array and populate it with pointers to the pieces of the allocated buffer; sglen should be set to the length of the array.

Drivers using the vmalloc() method need not (and cannot) concern themselves with buffer allocation at all; videobuf will handle those details. The same is normally true of contiguous-DMA drivers as well; videobuf will allocate the buffers (with dma_alloc_coherent()) when it sees fit. That means that these drivers may be trying to do high-order allocations at any time, an operation which is not always guaranteed to work. Some drivers play tricks by allocating DMA space at system boot time; videobuf does not currently play well with those drivers.

As of 2.6.31, contiguous-DMA drivers can work with a user-supplied buffer, as long as that buffer is physically contiguous. Normal user-space allocations will not meet that criterion, but buffers obtained from other kernel drivers, or those contained within huge pages, will work with these drivers.

Filling the buffers

The final part of a videobuf implementation has no direct callback - it' s the portion of the code which actually puts frame data into the buffers, usually in response to interrupts from the device. For all types of drivers, this process works approximately as follows:

- Obtain the next available buffer and make sure that somebody is actually waiting for it.
- Get a pointer to the memory and put video data there.
- Mark the buffer as done and wake up the process waiting for it.

Step (1) above is done by looking at the driver-managed list_head structure - the one which is filled in the buf_queue() callback. Because starting the engine and enqueueing buffers are done in separate steps, it' s possible for the engine to

be running without any buffers available - in the vmalloc() case especially. So the driver should be prepared for the list to be empty. It is equally possible that nobody is yet interested in the buffer; the driver should not remove it from the list or fill it until a process is waiting on it. That test can be done by examining the buffer's done field (a wait_queue_head_t structure) with `waitqueue_active()`.

A buffer's state should be set to VIDEOBUF_ACTIVE before being mapped for DMA; that ensures that the videobuf layer will not try to do anything with it while the device is transferring data.

For scatter/gather drivers, the needed memory pointers will be found in the scatterlist structure described above. Drivers using the vmalloc() method can get a memory pointer with:

```
void *videobuf_to_vmalloc(struct videobuf_buffer *buf);
```

For contiguous DMA drivers, the function to use is:

```
dma_addr_t videobuf_to_dma_contig(struct videobuf_buffer *buf);
```

The contiguous DMA API goes out of its way to hide the kernel-space address of the DMA buffer from drivers.

The final step is to set the size field of the relevant videobuf_buffer structure to the actual size of the captured image, set state to VIDEOBUF_DONE, then call wake_up() on the done queue. At this point, the buffer is owned by the videobuf layer and the driver should not touch it again.

Developers who are interested in more information can go into the relevant header files; there are a few low-level functions declared there which have not been talked about here. Note also that all of these calls are exported GPL-only, so they will not be available to non-GPL kernel modules.

53.1.15 V4L2 videobuf2 functions and data structures

enum **vb2_memory**
type of memory model used to make the buffers visible on userspace.

Constants

VB2_MEMORY_UNKNOWN Buffer status is unknown or it is not used yet on userspace.

VB2_MEMORY_MMAP The buffers are allocated by the Kernel and it is memory mapped via mmap() ioctl. This model is also used when the user is using the buffers via read() or write() system calls.

VB2_MEMORY_USERPTR The buffers was allocated in userspace and it is memory mapped via mmap() ioctl.

VB2_MEMORY_DMABUF The buffers are passed to userspace via DMA buffer.

struct **vb2_mem_ops**
memory handling/memory allocator operations.

Definition

```
struct vb2_mem_ops {
  void *(*alloc)(struct device *dev, unsigned long attrs,unsigned long␣
↪size,enum dma_data_direction dma_dir, gfp_t gfp_flags);
  void (*put)(void *buf_priv);
  struct dma_buf *(*get_dmabuf)(void *buf_priv, unsigned long flags);
  void *(*get_userptr)(struct device *dev, unsigned long vaddr,unsigned␣
↪long size, enum dma_data_direction dma_dir);
  void (*put_userptr)(void *buf_priv);
  void (*prepare)(void *buf_priv);
  void (*finish)(void *buf_priv);
  void *(*attach_dmabuf)(struct device *dev,struct dma_buf *dbuf,unsigned␣
↪long size, enum dma_data_direction dma_dir);
  void (*detach_dmabuf)(void *buf_priv);
  int (*map_dmabuf)(void *buf_priv);
  void (*unmap_dmabuf)(void *buf_priv);
  void *(*vaddr)(void *buf_priv);
  void *(*cookie)(void *buf_priv);
  unsigned int    (*num_users)(void *buf_priv);
  int (*mmap)(void *buf_priv, struct vm_area_struct *vma);
};
```

Members

alloc allocate video memory and, optionally, allocator private data, return ERR_PTR() on failure or a pointer to allocator private, per-buffer data on success; the returned private structure will then be passed as **buf_priv** argument to other ops in this structure. Additional gfp_flags to use when allocating the are also passed to this operation. These flags are from the gfp_flags field of vb2_queue. The size argument to this function shall be page aligned.

put inform the allocator that the buffer will no longer be used; usually will result in the allocator freeing the buffer (if no other users of this buffer are present); the **buf_priv** argument is the allocator private per-buffer structure previously returned from the alloc callback.

get_dmabuf acquire userspace memory for a hardware operation; used for DMABUF memory types.

get_userptr acquire userspace memory for a hardware operation; used for USERPTR memory types; vaddr is the address passed to the videobuf layer when queuing a video buffer of USERPTR type; should return an allocator private per-buffer structure associated with the buffer on success, ERR_PTR() on failure; the returned private structure will then be passed as **buf_priv** argument to other ops in this structure.

put_userptr inform the allocator that a USERPTR buffer will no longer be used.

prepare called every time the buffer is passed from userspace to the driver, useful for cache synchronisation, optional.

finish called every time the buffer is passed back from the driver to the userspace, also optional.

attach_dmabuf attach a shared `struct dma_buf` for a hardware operation; used for DMABUF memory types; dev is the alloc device dbuf is the shared dma_buf; returns ERR_PTR() on failure; allocator private per-buffer structure on success; this needs to be used for further accesses to the buffer.

detach_dmabuf inform the exporter of the buffer that the current DMABUF buffer is no longer used; the **buf_priv** argument is the allocator private per-buffer structure previously returned from the attach_dmabuf callback.

map_dmabuf request for access to the dmabuf from allocator; the allocator of dmabuf is informed that this driver is going to use the dmabuf.

unmap_dmabuf releases access control to the dmabuf - allocator is notified that this driver is done using the dmabuf for now.

vaddr return a kernel virtual address to a given memory buffer associated with the passed private structure or NULL if no such mapping exists.

cookie return allocator specific cookie for a given memory buffer associated with the passed private structure or NULL if not available.

num_users return the current number of users of a memory buffer; return 1 if the videobuf layer (or actually the driver using it) is the only user.

mmap setup a userspace mapping for a given memory buffer under the provided virtual memory region.

Description

Those operations are used by the videobuf2 core to implement the memory handling/memory allocators for each type of supported streaming I/O method.

Note:

1) Required ops for USERPTR types: get_userptr, put_userptr.
2) Required ops for MMAP types: alloc, put, num_users, mmap.
3) Required ops for read/write access types: alloc, put, num_users, vaddr.
4) Required ops for DMABUF types: attach_dmabuf, detach_dmabuf, map_dmabuf, unmap_dmabuf.

---

struct **vb2_plane**
plane information.

Definition

```
struct vb2_plane {
  void *mem_priv;
  struct dma_buf          *dbuf;
  unsigned int            dbuf_mapped;
  unsigned int            bytesused;
  unsigned int            length;
  unsigned int            min_length;
  union {
    unsigned int    offset;
    unsigned long   userptr;
    int fd;
  } m;
  unsigned int            data_offset;
};
```

Members

mem_priv private data with this plane.

dbuf dma_buf - shared buffer object.

dbuf_mapped flag to show whether dbuf is mapped or not

bytesused number of bytes occupied by data in the plane (payload).

length size of this plane (NOT the payload) in bytes.

min_length minimum required size of this plane (NOT the payload) in bytes. **length** is always greater or equal to **min_length**.

m Union with memtype-specific data.

m.offset when memory in the associated struct vb2_buffer is VB2_MEMORY_MMAP, equals the offset from the start of the device memory for this plane (or is a "cookie" that should be passed to mmap() called on the video node).

m.userptr when memory is VB2_MEMORY_USERPTR, a userspace pointer pointing to this plane.

m.fd when memory is VB2_MEMORY_DMABUF, a userspace file descriptor associated with this plane.

data_offset offset in the plane to the start of data; usually 0, unless there is a header in front of the data.

Description

Should contain enough information to be able to cover all the fields of `struct v4l2_plane` at videodev2.h.

enum **vb2_io_modes**
queue access methods.

Constants

VB2_MMAP driver supports MMAP with streaming API.

VB2_USERPTR driver supports USERPTR with streaming API.

VB2_READ driver supports read() style access.

VB2_WRITE driver supports write() style access.

VB2_DMABUF driver supports DMABUF with streaming API.

enum **vb2_buffer_state**
current video buffer state.

Constants

VB2_BUF_STATE_DEQUEUED buffer under userspace control.

VB2_BUF_STATE_IN_REQUEST buffer is queued in media request.

VB2_BUF_STATE_PREPARING buffer is being prepared in videobuf.

VB2_BUF_STATE_QUEUED buffer queued in videobuf, but not in driver.

VB2_BUF_STATE_ACTIVE buffer queued in driver and possibly used in a hardware operation.

VB2_BUF_STATE_DONE buffer returned from driver to videobuf, but not yet dequeued to userspace.

VB2_BUF_STATE_ERROR same as above, but the operation on the buffer has ended with an error, which will be reported to the userspace when it is dequeued.

struct **vb2_buffer**
represents a video buffer.

Definition

```
struct vb2_buffer {
  struct vb2_queue        *vb2_queue;
  unsigned int            index;
  unsigned int            type;
  unsigned int            memory;
  unsigned int            num_planes;
  u64 timestamp;
  struct media_request    *request;
  struct media_request_object     req_obj;
};
```

Members

vb2_queue pointer to `struct vb2_queue` with the queue to which this driver belongs.

index id number of the buffer.

type buffer type.

memory the method, in which the actual data is passed.

num_planes number of planes in the buffer on an internal driver queue.

timestamp frame timestamp in ns.

request the request this buffer is associated with.

req_obj used to bind this buffer to a request. This request object has a refcount.

struct **vb2_ops**
driver-specific callbacks.

Definition

```
struct vb2_ops {
  int (*queue_setup)(struct vb2_queue *q,unsigned int *num_buffers,␣
↪unsigned int *num_planes, unsigned int sizes[], struct device *alloc_
↪devs[]);
  void (*wait_prepare)(struct vb2_queue *q);
  void (*wait_finish)(struct vb2_queue *q);
  int (*buf_out_validate)(struct vb2_buffer *vb);
  int (*buf_init)(struct vb2_buffer *vb);
  int (*buf_prepare)(struct vb2_buffer *vb);
  void (*buf_finish)(struct vb2_buffer *vb);
  void (*buf_cleanup)(struct vb2_buffer *vb);
  int (*start_streaming)(struct vb2_queue *q, unsigned int count);
  void (*stop_streaming)(struct vb2_queue *q);
  void (*buf_queue)(struct vb2_buffer *vb);
```

(continues on next page)

(continued from previous page)

```
  void (*buf_request_complete)(struct vb2_buffer *vb);
};
```

Members

queue_setup called from VIDIOC_REQBUFS() and VIDIOC_CREATE_BUFS() handlers before memory allocation. It can be called twice: if the original number of requested buffers could not be allocated, then it will be called a second time with the actually allocated number of buffers to verify if that is OK. The driver should return the required number of buffers in *num_buffers, the required number of planes per buffer in *num_planes, the size of each plane should be set in the sizes[] array and optional per-plane allocator specific device in the alloc_devs[] array. When called from VIDIOC_REQBUFS(), *num_planes == 0, the driver has to use the currently configured format to determine the plane sizes and *num_buffers is the total number of buffers that are being allocated. When called from VIDIOC_CREATE_BUFS(), *num_planes != 0 and it describes the requested number of planes and sizes[] contains the requested plane sizes. In this case *num_buffers are being allocated additionally to q->num_buffers. If either *num_planes or the requested sizes are invalid callback must return `-EINVAL`.

wait_prepare release any locks taken while calling vb2 functions; it is called before an ioctl needs to wait for a new buffer to arrive; required to avoid a deadlock in blocking access type.

wait_finish reacquire all locks released in the previous callback; required to continue operation after sleeping while waiting for a new buffer to arrive.

buf_out_validate called when the output buffer is prepared or queued to a request; drivers can use this to validate userspace-provided information; this is required only for OUTPUT queues.

buf_init called once after allocating a buffer (in MMAP case) or after acquiring a new USERPTR buffer; drivers may perform additional buffer-related initialization; initialization failure (return != 0) will prevent queue setup from completing successfully; optional.

buf_prepare called every time the buffer is queued from userspace and from the VIDIOC_PREPARE_BUF() ioctl; drivers may perform any initialization required before each hardware operation in this callback; drivers can access/modify the buffer here as it is still synced for the CPU; drivers that support VIDIOC_CREATE_BUFS() must also validate the buffer size; if an error is returned, the buffer will not be queued in driver; optional.

buf_finish called before every dequeue of the buffer back to userspace; the buffer is synced for the CPU, so drivers can access/modify the buffer contents; drivers may perform any operations required before userspace accesses the buffer; optional. The buffer state can be one of the following: `DONE` and `ERROR` occur while streaming is in progress, and the `PREPARED` state occurs when the queue has been canceled and all pending buffers are being returned to their default `DEQUEUED` state. Typically you only have to do something if the state is `VB2_BUF_STATE_DONE`, since in all other cases the buffer contents will be ignored anyway.

buf_cleanup called once before the buffer is freed; drivers may perform any additional cleanup; optional.

start_streaming called once to enter 'streaming' state; the driver may receive buffers with **buf_queue** callback before **start_streaming** is called; the driver gets the number of already queued buffers in count parameter; driver can return an error if hardware fails, in that case all buffers that have been already given by the **buf_queue** callback are to be returned by the driver by calling `vb2_buffer_done()` with `VB2_BUF_STATE_QUEUED`. If you need a minimum number of buffers before you can start streaming, then set `vb2_queue->min_buffers_needed`. If that is non-zero then **start_streaming** won't be called until at least that many buffers have been queued up by userspace.

stop_streaming called when 'streaming' state must be disabled; driver should stop any DMA transactions or wait until they finish and give back all buffers it got from `buf_queue` callback by calling `vb2_buffer_done()` with either `VB2_BUF_STATE_DONE` or `VB2_BUF_STATE_ERROR`; may use `vb2_wait_for_all_buffers()` function

buf_queue passes buffer vb to the driver; driver may start hardware operation on this buffer; driver should give the buffer back by calling `vb2_buffer_done()` function; it is always called after calling VIDIOC_STREAMON() ioctl; might be called before **start_streaming** callback if user pre-queued buffers before calling VIDIOC_STREAMON().

buf_request_complete a buffer that was never queued to the driver but is associated with a queued request was canceled. The driver will have to mark associated objects in the request as completed; required if requests are supported.

Description

These operations are not called from interrupt context except where mentioned specifically.

struct **vb2_buf_ops**
driver-specific callbacks.

Definition

```
struct vb2_buf_ops {
  int (*verify_planes_array)(struct vb2_buffer *vb, const void *pb);
  void (*init_buffer)(struct vb2_buffer *vb);
  void (*fill_user_buffer)(struct vb2_buffer *vb, void *pb);
  int (*fill_vb2_buffer)(struct vb2_buffer *vb, struct vb2_plane *planes);
  void (*copy_timestamp)(struct vb2_buffer *vb, const void *pb);
};
```

Members

verify_planes_array Verify that a given user space structure contains enough planes for the buffer. This is called for each dequeued buffer.

init_buffer given a `vb2_buffer` initialize the extra data after struct vb2_buffer. For V4L2 this is a `struct vb2_v4l2_buffer`.

fill_user_buffer given a vb2_buffer fill in the userspace structure. For V4L2 this is a struct v4l2_buffer.

fill_vb2_buffer given a userspace structure, fill in the vb2_buffer. If the userspace structure is invalid, then this op will return an error.

copy_timestamp copy the timestamp from a userspace structure to the struct vb2_buffer.

struct **vb2_queue**
a videobuf queue.

Definition

```
struct vb2_queue {
  unsigned int                    type;
  unsigned int                    io_modes;
  struct device                   *dev;
  unsigned long                   dma_attrs;
  unsigned bidirectional:1;
  unsigned fileio_read_once:1;
  unsigned fileio_write_immediately:1;
  unsigned allow_zero_bytesused:1;
  unsigned quirk_poll_must_check_waiting_for_buffers:1;
  unsigned supports_requests:1;
  unsigned requires_requests:1;
  unsigned uses_qbuf:1;
  unsigned uses_requests:1;
  struct mutex                    *lock;
  void *owner;
  const struct vb2_ops            *ops;
  const struct vb2_mem_ops        *mem_ops;
  const struct vb2_buf_ops        *buf_ops;
  void *drv_priv;
  u32 subsystem_flags;
  unsigned int                    buf_struct_size;
  u32 timestamp_flags;
  gfp_t gfp_flags;
  u32 min_buffers_needed;
  struct device                   *alloc_devs[VB2_MAX_PLANES];
};
```

Members

type private buffer type whose content is defined by the vb2-core caller. For example, for V4L2, it should match the types defined on enum v4l2_buf_type.

io_modes supported io methods (see enum vb2_io_modes).

dev device to use for the default allocation context if the driver doesn't fill in the **alloc_devs** array.

dma_attrs DMA attributes to use for the DMA.

bidirectional when this flag is set the DMA direction for the buffers of this queue will be overridden with DMA_BIDIRECTIONAL direction. This is useful in cases where the hardware (firmware) writes to a buffer which is mapped as read (DMA_TO_DEVICE), or reads from buffer which is mapped for write (DMA_FROM_DEVICE) in order to satisfy some internal hardware restrictions

or adds a padding needed by the processing algorithm. In case the DMA mapping is not bidirectional but the hardware (firmware) trying to access the buffer (in the opposite direction) this could lead to an IOMMU protection faults.

fileio_read_once report EOF after reading the first buffer

fileio_write_immediately queue buffer after each write() call

allow_zero_bytesused allow bytesused == 0 to be passed to the driver

quirk_poll_must_check_waiting_for_buffers Return EPOLLERR at poll when QBUF has not been called. This is a vb1 idiom that has been adopted also by vb2.

supports_requests this queue supports the Request API.

requires_requests this queue requires the Request API. If this is set to 1, then supports_requests must be set to 1 as well.

uses_qbuf qbuf was used directly for this queue. Set to 1 the first time this is called. Set to 0 when the queue is canceled. If this is 1, then you cannot queue buffers from a request.

uses_requests requests are used for this queue. Set to 1 the first time a request is queued. Set to 0 when the queue is canceled. If this is 1, then you cannot queue buffers directly.

lock pointer to a mutex that protects the `struct vb2_queue`. The driver can set this to a mutex to let the v4l2 core serialize the queuing ioctls. If the driver wants to handle locking itself, then this should be set to NULL. This lock is not used by the videobuf2 core API.

owner The filehandle that 'owns' the buffers, i.e. the filehandle that called reqbufs, create_buffers or started fileio. This field is not used by the videobuf2 core API, but it allows drivers to easily associate an owner filehandle with the queue.

ops driver-specific callbacks

mem_ops memory allocator specific callbacks

buf_ops callbacks to deliver buffer information. between user-space and kernel-space.

drv_priv driver private data.

subsystem_flags Flags specific to the subsystem (V4L2/DVB/etc.). Not used by the vb2 core.

buf_struct_size size of the driver-specific buffer structure; "0" indicates the driver doesn't want to use a custom buffer structure type. In that case a subsystem-specific struct will be used (in the case of V4L2 that is `sizeof(struct vb2_v4l2_buffer)`). The first field of the driver-specific buffer structure must be the subsystem-specific struct (vb2_v4l2_buffer in the case of V4L2).

timestamp_flags Timestamp flags; `V4L2_BUF_FLAG_TIMESTAMP_*` and `V4L2_BUF_FLAG_TSTAMP_SRC_*`

gfp_flags additional gfp flags used when allocating the buffers. Typically this is 0, but it may be e.g. GFP_DMA or __GFP_DMA32 to force the buffer allocation to a specific memory zone.

min_buffers_needed the minimum number of buffers needed before **start_streaming** can be called. Used when a DMA engine cannot be started unless at least this number of buffers have been queued into the driver.

alloc_devs struct device memory type/allocator-specific per-plane device

void * **vb2_plane_vaddr**(struct vb2_buffer * vb, unsigned int plane_no)
Return a kernel virtual address of a given plane.

Parameters

struct vb2_buffer * vb pointer to struct vb2_buffer to which the plane in question belongs to.

unsigned int plane_no plane number for which the address is to be returned.

Description

This function returns a kernel virtual address of a given plane if such a mapping exist, NULL otherwise.

void * **vb2_plane_cookie**(struct vb2_buffer * vb, unsigned int plane_no)
Return allocator specific cookie for the given plane.

Parameters

struct vb2_buffer * vb pointer to struct vb2_buffer to which the plane in question belongs to.

unsigned int plane_no plane number for which the cookie is to be returned.

Description

This function returns an allocator specific cookie for a given plane if available, NULL otherwise. The allocator should provide some simple static inline function, which would convert this cookie to the allocator specific type that can be used directly by the driver to access the buffer. This can be for example physical address, pointer to scatter list or IOMMU mapping.

void **vb2_buffer_done**(struct vb2_buffer * vb, enum vb2_buffer_state state)
inform videobuf that an operation on a buffer is finished.

Parameters

struct vb2_buffer * vb pointer to struct vb2_buffer to be used.

enum vb2_buffer_state state state of the buffer, as defined by enum vb2_buffer_state. Either VB2_BUF_STATE_DONE if the operation finished successfully, VB2_BUF_STATE_ERROR if the operation finished with an error or VB2_BUF_STATE_QUEUED.

Description

This function should be called by the driver after a hardware operation on a buffer is finished and the buffer may be returned to userspace. The driver cannot use this buffer anymore until it is queued back to it by videobuf by the means of

vb2_ops->buf_queue callback. Only buffers previously queued to the driver by vb2_ops->buf_queue can be passed to this function.

While streaming a buffer can only be returned in state DONE or ERROR. The vb2_ops->start_streaming op can also return them in case the DMA engine cannot be started for some reason. In that case the buffers should be returned with state QUEUED to put them back into the queue.

void **vb2_discard_done**(struct vb2_queue * q)
discard all buffers marked as DONE.

Parameters

struct vb2_queue * q pointer to struct vb2_queue with videobuf2 queue.

Description

This function is intended to be used with suspend/resume operations. It discards all 'done' buffers as they would be too old to be requested after resume.

Drivers must stop the hardware and synchronize with interrupt handlers and/or delayed works before calling this function to make sure no buffer will be touched by the driver and/or hardware.

int **vb2_wait_for_all_buffers**(struct vb2_queue * q)
wait until all buffers are given back to vb2.

Parameters

struct vb2_queue * q pointer to struct vb2_queue with videobuf2 queue.

Description

This function will wait until all buffers that have been given to the driver by vb2_ops->buf_queue are given back to vb2 with vb2_buffer_done(). It doesn't call vb2_ops->wait_prepare/vb2_ops->wait_finish pair. It is intended to be called with all locks taken, for example from vb2_ops->stop_streaming callback.

void **vb2_core_querybuf**(struct vb2_queue * q, unsigned int index, void * pb)
query video buffer information.

Parameters

struct vb2_queue * q pointer to struct vb2_queue with videobuf2 queue.

unsigned int index id number of the buffer.

void * pb buffer struct passed from userspace.

Description

Videobuf2 core helper to implement VIDIOC_QUERYBUF() operation. It is called internally by VB2 by an API-specific handler, like videobuf2-v4l2.h.

The passed buffer should have been verified.

This function fills the relevant information for the userspace.

Return

returns zero on success; an error code otherwise.

int **vb2_core_reqbufs**(struct vb2_queue * q, enum vb2_memory memory, unsigned int * count)

Initiate streaming.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

enum vb2_memory memory memory type, as defined by `enum vb2_memory`.

unsigned int * count requested buffer count.

Description

Videobuf2 core helper to implement VIDIOC_REQBUF() operation. It is called internally by VB2 by an API-specific handler, like `videobuf2-v4l2.h`.

This function:

1) verifies streaming parameters passed from the userspace;
2) sets up the queue;
3) negotiates number of buffers and planes per buffer with the driver to be used during streaming;
4) allocates internal buffer structures (`struct vb2_buffer`), according to the agreed parameters;
5) for MMAP memory type, allocates actual video memory, using the memory handling/allocation routines provided during queue initialization.

If req->count is 0, all the memory will be freed instead.

If the queue has been allocated previously by a previous `vb2_core_reqbufs()` call and the queue is not busy, memory will be reallocated.

Return

returns zero on success; an error code otherwise.

int **vb2_core_create_bufs**(struct vb2_queue * q, enum vb2_memory memory, unsigned int * count, unsigned int requested_planes, const unsigned int requested_sizes)

Allocate buffers and any required auxiliary structs

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

enum vb2_memory memory memory type, as defined by `enum vb2_memory`.

unsigned int * count requested buffer count.

unsigned int requested_planes number of planes requested.

const unsigned int requested_sizes array with the size of the planes.

Description

Videobuf2 core helper to implement VIDIOC_CREATE_BUFS() operation. It is called internally by VB2 by an API-specific handler, like `videobuf2-v4l2.h`.

This function:

1) verifies parameter sanity;
2) calls the `vb2_ops->queue_setup` queue operation;
3) performs any necessary memory allocations.

Return

returns zero on success; an error code otherwise.

int **vb2_core_prepare_buf**(struct vb2_queue * q, unsigned int index, void * pb)
Pass ownership of a buffer from userspace to the kernel.

Parameters

`struct vb2_queue * q` pointer to `struct vb2_queue` with videobuf2 queue.

`unsigned int index` id number of the buffer.

`void * pb` buffer structure passed from userspace to `v4l2_ioctl_ops->vidioc_prepare_buf` handler in driver.

Description

Videobuf2 core helper to implement VIDIOC_PREPARE_BUF() operation. It is called internally by VB2 by an API-specific handler, like `videobuf2-v4l2.h`.

The passed buffer should have been verified.

This function calls vb2_ops->buf_prepare callback in the driver (if provided), in which driver-specific buffer initialization can be performed.

Return

returns zero on success; an error code otherwise.

int **vb2_core_qbuf**(struct vb2_queue * q, unsigned int index, void * pb, struct media_request * req)
Queue a buffer from userspace

Parameters

`struct vb2_queue * q` pointer to `struct vb2_queue` with videobuf2 queue.

`unsigned int index` id number of the buffer

`void * pb` buffer structure passed from userspace to v4l2_ioctl_ops->vidioc_qbuf handler in driver

`struct media_request * req` pointer to `struct media_request`, may be NULL.

Description

Videobuf2 core helper to implement VIDIOC_QBUF() operation. It is called internally by VB2 by an API-specific handler, like `videobuf2-v4l2.h`.

This function:

1) If **req** is non-NULL, then the buffer will be bound to this media request and it returns. The buffer will be prepared and queued to the driver (i.e. the next two steps) when the request itself is queued.

2) if necessary, calls `vb2_ops->buf_prepare` callback in the driver (if provided), in which driver-specific buffer initialization can be performed;

3) if streaming is on, queues the buffer in driver by the means of `vb2_ops->buf_queue` callback for processing.

Return

returns zero on success; an error code otherwise.

int **vb2_core_dqbuf**(struct vb2_queue * q, unsigned int * pindex, void * pb, bool nonblocking)
Dequeue a buffer to the userspace

Parameters

`struct vb2_queue * q` pointer to `struct vb2_queue` with videobuf2 queue

`unsigned int * pindex` pointer to the buffer index. May be NULL

`void * pb` buffer structure passed from userspace to v4l2_ioctl_ops->vidioc_dqbuf handler in driver.

`bool nonblocking` if true, this call will not sleep waiting for a buffer if no buffers ready for dequeuing are present. Normally the driver would be passing (file->f_flags & O_NONBLOCK) here.

Description

Videobuf2 core helper to implement VIDIOC_DQBUF() operation. It is called internally by VB2 by an API-specific handler, like `videobuf2-v4l2.h`.

This function:

1) calls buf_finish callback in the driver (if provided), in which driver can perform any additional operations that may be required before returning the buffer to userspace, such as cache sync,

2) the buffer struct members are filled with relevant information for the userspace.

Return

returns zero on success; an error code otherwise.

int **vb2_core_streamon**(struct vb2_queue * q, unsigned int type)
Implements VB2 stream ON logic

Parameters

`struct vb2_queue * q` pointer to `struct vb2_queue` with videobuf2 queue

`unsigned int type` type of the queue to be started. For V4L2, this is defined by `enum v4l2_buf_type` type.

Description

Videobuf2 core helper to implement VIDIOC_STREAMON() operation. It is called internally by VB2 by an API-specific handler, like `videobuf2-v4l2.h`.

Return

returns zero on success; an error code otherwise.

int **vb2_core_streamoff**(struct vb2_queue * q, unsigned int type)
Implements VB2 stream OFF logic

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue

unsigned int type type of the queue to be started. For V4L2, this is defined by `enum v4l2_buf_type` type.

Description

Videobuf2 core helper to implement VIDIOC_STREAMOFF() operation. It is called internally by VB2 by an API-specific handler, like `videobuf2-v4l2.h`.

Return

returns zero on success; an error code otherwise.

int **vb2_core_expbuf**(struct vb2_queue * q, int * fd, unsigned int type, unsigned int index, unsigned int plane, unsigned int flags)
Export a buffer as a file descriptor.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

int * fd pointer to the file descriptor associated with DMABUF (set by driver).

unsigned int type buffer type.

unsigned int index id number of the buffer.

unsigned int plane index of the plane to be exported, 0 for single plane queues

unsigned int flags file flags for newly created file, as defined at include/uapi/asm-generic/fcntl.h. Currently, the only used flag is `O_CLOEXEC`. is supported, refer to manual of open syscall for more details.

Description

Videobuf2 core helper to implement VIDIOC_EXPBUF() operation. It is called internally by VB2 by an API-specific handler, like `videobuf2-v4l2.h`.

Return

returns zero on success; an error code otherwise.

int **vb2_core_queue_init**(struct vb2_queue * q)
initialize a videobuf2 queue

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue. This structure should be allocated in driver

Description

The `vb2_queue` structure should be allocated by the driver. The driver is responsible of clearing it's content and setting initial values for some required entries before calling this function.

Note: The following fields at **q** should be set before calling this function: `vb2_queue->ops`, `vb2_queue->mem_ops`, `vb2_queue->type`.

void **vb2_core_queue_release**(struct vb2_queue * q)
stop streaming, release the queue and free memory

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

Description

This function stops streaming and performs necessary clean ups, including freeing video buffer memory. The driver is responsible for freeing the `struct vb2_queue` itself.

void **vb2_queue_error**(struct vb2_queue * q)
signal a fatal error on the queue

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

Description

Flag that a fatal unrecoverable error has occurred and wake up all processes waiting on the queue. Polling will now set `EPOLLERR` and queuing and dequeuing buffers will return `-EIO`.

The error flag will be cleared when canceling the queue, either from `vb2_streamoff()` or `vb2_queue_release()`. Drivers should thus not call this function before starting the stream, otherwise the error flag will remain set until the queue is released when closing the device node.

int **vb2_mmap**(struct vb2_queue * q, struct vm_area_struct * vma)
map video buffers into application address space.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct vm_area_struct * vma pointer to `struct vm_area_struct` with the vma passed to the mmap file operation handler in the driver.

Description

Should be called from mmap file operation handler of a driver. This function maps one plane of one of the available video buffers to userspace. To map whole video memory allocated on reqbufs, this function has to be called once per each plane per each buffer previously allocated.

When the userspace application calls mmap, it passes to it an offset returned to it earlier by the means of `v4l2_ioctl_ops->vidioc_querybuf` handler. That offset acts as a “cookie”, which is then used to identify the plane to be mapped.

This function finds a plane with a matching offset and a mapping is performed by the means of a provided memory operation.

The return values from this function are intended to be directly returned from the mmap handler in driver.

unsigned long **vb2_get_unmapped_area**(struct vb2_queue * q, unsigned long addr, unsigned long len, unsigned long pgoff, unsigned long flags)
map video buffers into application address space.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

unsigned long addr memory address.

unsigned long len buffer size.

unsigned long pgoff page offset.

unsigned long flags memory flags.

Description

This function is used in noMMU platforms to propose address mapping for a given buffer. It's intended to be used as a handler for the `file_operations->get_unmapped_area` operation.

This is called by the mmap() syscall routines will call this to get a proposed address for the mapping, when `!CONFIG_MMU`.

__poll_t **vb2_core_poll**(struct vb2_queue * q, struct file * file, poll_table * wait)
implements poll syscall() logic.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct file * file `struct file` argument passed to the poll file operation handler.

poll_table * wait `poll_table` wait argument passed to the poll file operation handler.

Description

This function implements poll file operation handler for a driver. For CAPTURE queues, if a buffer is ready to be dequeued, the userspace will be informed that the file descriptor of a video device is available for reading. For OUTPUT queues, if a buffer is ready to be dequeued, the file descriptor will be reported as available for writing.

The return values from this function are intended to be directly returned from poll handler in driver.

size_t **vb2_read**(struct vb2_queue * q, char __user * data, size_t count, loff_t * ppos, int nonblock)
implements read() syscall logic.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

char __user * data pointed to target userspace buffer

size_t count number of bytes to read

loff_t * ppos file handle position tracking pointer

int nonblock mode selector (1 means blocking calls, 0 means nonblocking)

size_t **vb2_write**(struct vb2_queue * q, const char __user * data, size_t count, loff_t * ppos, int nonblock)
implements write() syscall logic.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

const char __user * data pointed to target userspace buffer

size_t count number of bytes to write

loff_t * ppos file handle position tracking pointer

int nonblock mode selector (1 means blocking calls, 0 means nonblocking)

vb2_thread_fnc
Typedef: callback function for use with vb2_thread.

Syntax

```
int vb2_thread_fnc (struct vb2_buffer * vb, void * priv);
```

Parameters

struct vb2_buffer * vb pointer to struct `vb2_buffer`.

void * priv pointer to a private data.

Description

This is called whenever a buffer is dequeued in the thread.

int **vb2_thread_start**(struct vb2_queue * q, vb2_thread_fnc fnc, void * priv, const char * thread_name)
start a thread for the given queue.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

vb2_thread_fnc fnc `vb2_thread_fnc` callback function.

void * priv priv pointer passed to the callback function.

const char * thread_name the name of the thread. This will be prefixed with “vb2- “.

Description

This starts a thread that will queue and dequeue until an error occurs or `vb2_thread_stop()` is called.

Attention: This function should not be used for anything else but the videobuf2-dvb support. If you think you have another good use-case for this, then please contact the linux-media mailing list first.

int **vb2_thread_stop**(struct vb2_queue * q)
stop the thread for the given queue.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

bool **vb2_is_streaming**(struct vb2_queue * q)
return streaming status of the queue.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

bool **vb2_fileio_is_active**(struct vb2_queue * q)
return true if fileio is active.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

Description

This returns true if read() or write() is used to stream the data as opposed to stream I/O. This is almost never an important distinction, except in rare cases. One such case is that using read() or write() to stream a format using `V4L2_FIELD_ALTERNATE` is not allowed since there is no way you can pass the field information of each buffer to/from userspace. A driver that supports this field format should check for this in the `vb2_ops->queue_setup` op and reject it if this function returns true.

bool **vb2_is_busy**(struct vb2_queue * q)
return busy status of the queue.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

Description

This function checks if queue has any buffers allocated.

void * **vb2_get_drv_priv**(struct vb2_queue * q)
return driver private data associated with the queue.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

void **vb2_set_plane_payload**(struct vb2_buffer * vb, unsigned int plane_no,
unsigned long size)
set bytesused for the plane **plane_no**.

Parameters

struct vb2_buffer * vb pointer to `struct vb2_buffer` to which the plane in question belongs to.

unsigned int plane_no plane number for which payload should be set.

unsigned long size payload in bytes.

unsigned long **vb2_get_plane_payload**(struct vb2_buffer * vb, unsigned int plane_no)
get bytesused for the plane plane_no

Parameters

struct vb2_buffer * vb pointer to `struct vb2_buffer` to which the plane in question belongs to.

unsigned int plane_no plane number for which payload should be set.

unsigned long **vb2_plane_size**(struct vb2_buffer * vb, unsigned int plane_no)
return plane size in bytes.

Parameters

struct vb2_buffer * vb pointer to `struct vb2_buffer` to which the plane in question belongs to.

unsigned int plane_no plane number for which size should be returned.

bool **vb2_start_streaming_called**(struct vb2_queue * q)
return streaming status of driver.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

void **vb2_clear_last_buffer_dequeued**(struct vb2_queue * q)
clear last buffer dequeued flag of queue.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct vb2_buffer * **vb2_get_buffer**(struct vb2_queue * q, unsigned int index)
get a buffer from a queue

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

unsigned int index buffer index

Description

This function obtains a buffer from a queue, by its index. Keep in mind that there is no refcounting involved in this operation, so the buffer lifetime should be taken into consideration.

bool **vb2_buffer_in_use**(struct vb2_queue * q, struct vb2_buffer * vb)
return true if the buffer is in use and the queue cannot be freed (by the means of VIDIOC_REQBUFS(0)) call.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct vb2_buffer * vb buffer for which plane size should be returned.

int **vb2_verify_memory_type**(struct vb2_queue * q, enum vb2_memory memory, unsigned int type)
Check whether the memory type and buffer type passed to a buffer operation are compatible with the queue.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

enum vb2_memory memory memory model, as defined by enum `vb2_memory`.

unsigned int type private buffer type whose content is defined by the vb2-core caller. For example, for V4L2, it should match the types defined on enum `v4l2_buf_type`.

bool **vb2_request_object_is_buffer**(struct media_request_object * obj)
return true if the object is a buffer

Parameters

struct media_request_object * obj the request object.

unsigned int **vb2_request_buffer_cnt**(struct media_request * req)
return the number of buffers in the request

Parameters

struct media_request * req the request.

struct **vb2_v4l2_buffer**
video buffer information for v4l2.

Definition

```
struct vb2_v4l2_buffer {
  struct vb2_buffer       vb2_buf;
  __u32 flags;
  __u32 field;
  struct v4l2_timecode    timecode;
  __u32 sequence;
  __s32 request_fd;
  bool is_held;
  struct vb2_plane        planes[VB2_MAX_PLANES];
};
```

Members

vb2_buf embedded struct `vb2_buffer`.

flags buffer informational flags.

field field order of the image in the buffer, as defined by `enum v4l2_field`.

timecode frame timecode.

sequence sequence count of this frame.

request_fd the request_fd associated with this buffer

is_held if true, then this capture buffer was held

planes plane information (userptr/fd, length, bytesused, data_offset).

Description

Should contain enough information to be able to cover all the fields of `struct v4l2_buffer` at `videodev2.h`.

int **vb2_find_timestamp**(const struct vb2_queue * q, u64 timestamp, unsigned int start_idx)
Find buffer with given timestamp in the queue

Parameters

const struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

u64 timestamp the timestamp to find.

unsigned int start_idx the start index (usually 0) in the buffer array to start searching from. Note that there may be multiple buffers with the same timestamp value, so you can restart the search by setting **start_idx** to the previously found index + 1.

Description

Returns the buffer index of the buffer with the given **timestamp**, or -1 if no buffer with **timestamp** was found.

int **vb2_reqbufs**(struct vb2_queue * q, struct v4l2_requestbuffers * req)
Wrapper for `vb2_core_reqbufs()` that also verifies the memory and type values.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct v4l2_requestbuffers * req `struct v4l2_requestbuffers` passed from userspace to `v4l2_ioctl_ops->vidioc_reqbufs` handler in driver.

int **vb2_create_bufs**(struct vb2_queue * q, struct v4l2_create_buffers * create)
Wrapper for `vb2_core_create_bufs()` that also verifies the memory and type values.

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct v4l2_create_buffers * create creation parameters, passed from userspace to `v4l2_ioctl_ops->vidioc_create_bufs` handler in driver

int **vb2_prepare_buf**(struct vb2_queue * q, struct media_device * mdev, struct v4l2_buffer * b)
Pass ownership of a buffer from userspace to the kernel

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct media_device * mdev pointer to `struct media_device`, may be NULL.

struct v4l2_buffer * b buffer structure passed from userspace to `v4l2_ioctl_ops->vidioc_prepare_buf` handler in driver

Description

Should be called from `v4l2_ioctl_ops->vidioc_prepare_buf` ioctl handler of a driver.

This function:

1) verifies the passed buffer,
2) calls `vb2_ops->buf_prepare` callback in the driver (if provided), in which driver-specific buffer initialization can be performed.
3) if **b->request_fd** is non-zero and **mdev->ops->req_queue** is set, then bind the prepared buffer to the request.

The return values from this function are intended to be directly returned from `v4l2_ioctl_ops->vidioc_prepare_buf` handler in driver.

int **vb2_qbuf**(struct vb2_queue * q, struct media_device * mdev, struct v4l2_buffer * b)
Queue a buffer from userspace

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct media_device * mdev pointer to `struct media_device`, may be NULL.

struct v4l2_buffer * b buffer structure passed from userspace to `v4l2_ioctl_ops->vidioc_qbuf` handler in driver

Description

Should be called from `v4l2_ioctl_ops->vidioc_qbuf` handler of a driver.

This function:

1) verifies the passed buffer;
2) if **b->request_fd** is non-zero and **mdev->ops->req_queue** is set, then bind the buffer to the request.
3) if necessary, calls `vb2_ops->buf_prepare` callback in the driver (if provided), in which driver-specific buffer initialization can be performed;
4) if streaming is on, queues the buffer in driver by the means of `vb2_ops->buf_queue` callback for processing.

The return values from this function are intended to be directly returned from `v4l2_ioctl_ops->vidioc_qbuf` handler in driver.

int **vb2_expbuf**(struct vb2_queue * q, struct v4l2_exportbuffer * eb)
Export a buffer as a file descriptor

Parameters

struct vb2_queue * q pointer to `struct vb2_queue` with videobuf2 queue.

struct v4l2_exportbuffer * eb export buffer structure passed from userspace to `v4l2_ioctl_ops->vidioc_expbuf` handler in driver

Description

The return values from this function are intended to be directly returned from `v4l2_ioctl_ops->vidioc_expbuf` handler in driver.

int **vb2_dqbuf**(struct vb2_queue * q, struct v4l2_buffer * b, bool nonblocking)
Dequeue a buffer to the userspace

Parameters

`struct vb2_queue * q` pointer to `struct vb2_queue` with videobuf2 queue.

`struct v4l2_buffer * b` buffer structure passed from userspace to `v4l2_ioctl_ops->vidioc_dqbuf` handler in driver

`bool nonblocking` if true, this call will not sleep waiting for a buffer if no buffers ready for dequeuing are present. Normally the driver would be passing (`file->f_flags` & `O_NONBLOCK`) here

Description

Should be called from `v4l2_ioctl_ops->vidioc_dqbuf` ioctl handler of a driver.

This function:

1) verifies the passed buffer;
2) calls `vb2_ops->buf_finish` callback in the driver (if provided), in which driver can perform any additional operations that may be required before returning the buffer to userspace, such as cache sync;
3) the buffer struct members are filled with relevant information for the userspace.

The return values from this function are intended to be directly returned from `v4l2_ioctl_ops->vidioc_dqbuf` handler in driver.

int **vb2_streamon**(struct vb2_queue * q, enum v4l2_buf_type type)
start streaming

Parameters

`struct vb2_queue * q` pointer to `struct vb2_queue` with videobuf2 queue.

`enum v4l2_buf_type type` type argument passed from userspace to vidioc_streamon handler, as defined by `enum v4l2_buf_type`.

Description

Should be called from `v4l2_ioctl_ops->vidioc_streamon` handler of a driver.

This function:

1) verifies current state
2) passes any previously queued buffers to the driver and starts streaming

The return values from this function are intended to be directly returned from `v4l2_ioctl_ops->vidioc_streamon` handler in the driver.

int **vb2_streamoff**(struct vb2_queue * q, enum v4l2_buf_type type)
stop streaming

Parameters

struct vb2_queue * q pointer to struct vb2_queue with videobuf2 queue.

enum v4l2_buf_type type type argument passed from userspace to vidioc_streamoff handler

Description

Should be called from vidioc_streamoff handler of a driver.

This function:

1) verifies current state,

2) stop streaming and dequeues any queued buffers, including those previously passed to the driver (after waiting for the driver to finish).

This call can be used for pausing playback. The return values from this function are intended to be directly returned from vidioc_streamoff handler in the driver

int **vb2_queue_init**(struct vb2_queue * q)
initialize a videobuf2 queue

Parameters

struct vb2_queue * q pointer to struct vb2_queue with videobuf2 queue.

Description

The vb2_queue structure should be allocated by the driver. The driver is responsible of clearing it's content and setting initial values for some required entries before calling this function. q->ops, q->mem_ops, q->type and q->io_modes are mandatory. Please refer to the struct vb2_queue description in include/media/videobuf2-core.h for more information.

void **vb2_queue_release**(struct vb2_queue * q)
stop streaming, release the queue and free memory

Parameters

struct vb2_queue * q pointer to struct vb2_queue with videobuf2 queue.

Description

This function stops streaming and performs necessary clean ups, including freeing video buffer memory. The driver is responsible for freeing the vb2_queue structure itself.

__poll_t **vb2_poll**(struct vb2_queue * q, struct file * file, poll_table * wait)
implements poll userspace operation

Parameters

struct vb2_queue * q pointer to struct vb2_queue with videobuf2 queue.

struct file * file file argument passed to the poll file operation handler

poll_table * wait wait argument passed to the poll file operation handler

Description

This function implements poll file operation handler for a driver. For CAPTURE queues, if a buffer is ready to be dequeued, the userspace will be informed that the file descriptor of a video device is available for reading. For OUTPUT queues,

if a buffer is ready to be dequeued, the file descriptor will be reported as available for writing.

If the driver uses struct v4l2_fh, then `vb2_poll()` will also check for any pending events.

The return values from this function are intended to be directly returned from poll handler in driver.

void **vb2_ops_wait_prepare**(struct vb2_queue * vq)
helper function to lock a struct `vb2_queue`

Parameters

`struct vb2_queue * vq` pointer to `struct vb2_queue`

Description

..note:: only use if vq->lock is non-NULL.

void **vb2_ops_wait_finish**(struct vb2_queue * vq)
helper function to unlock a struct `vb2_queue`

Parameters

`struct vb2_queue * vq` pointer to `struct vb2_queue`

Description

..note:: only use if vq->lock is non-NULL.

struct **vb2_vmarea_handler**
common vma refcount tracking handler.

Definition

```
struct vb2_vmarea_handler {
  refcount_t *refcount;
  void (*put)(void *arg);
  void *arg;
};
```

Members

`refcount` pointer to `refcount_t` entry in the buffer.

`put` callback to function that decreases buffer refcount.

`arg` argument for **put** callback.

53.1.16 V4L2 clocks

Attention: This is a temporary API and it shall be replaced by the generic clock API, when the latter becomes widely available.

Many subdevices, like camera sensors, TV decoders and encoders, need a clock signal to be supplied by the system. Often this clock is supplied by the respective bridge device. The Linux kernel provides a Common Clock Framework for this

purpose. However, it is not (yet) available on all architectures. Besides, the nature of the multi-functional (clock, data + synchronisation, I2C control) connection of subdevices to the system might impose special requirements on the clock API usage. E.g. V4L2 has to support clock provider driver unregistration while a subdevice driver is holding a reference to the clock. For these reasons a V4L2 clock helper API has been developed and is provided to bridge and subdevice drivers.

The API consists of two parts: two functions to register and unregister a V4L2 clock source: v4l2_clk_register() and v4l2_clk_unregister() and calls to control a clock object, similar to the respective generic clock API calls: v4l2_clk_get(), v4l2_clk_put(), v4l2_clk_enable(), v4l2_clk_disable(), v4l2_clk_get_rate(), and v4l2_clk_set_rate(). Clock suppliers have to provide clock operations that will be called when clock users invoke respective API methods.

It is expected that once the CCF becomes available on all relevant architectures this API will be removed.

53.1.17 V4L2 DV Timings functions

struct v4l2_fract **v4l2_calc_timeperframe**(const struct v4l2_dv_timings * t)

helper function to calculate timeperframe based v4l2_dv_timings fields.

Parameters

const struct v4l2_dv_timings * t Timings for the video mode.

Description

Calculates the expected timeperframe using the pixel clock value and horizontal/vertical measures. This means that v4l2_dv_timings structure must be correctly and fully filled.

v4l2_check_dv_timings_fnc

Typedef: timings check callback

Syntax

```
bool v4l2_check_dv_timings_fnc (const struct v4l2_dv_timings
* t, void * handle);
```

Parameters

const struct v4l2_dv_timings * t the v4l2_dv_timings struct.

void * handle a handle from the driver.

Description

Returns true if the given timings are valid.

bool **v4l2_valid_dv_timings**(const struct v4l2_dv_timings * t, const struct v4l2_dv_timings_cap * cap, v4l2_check_dv_timings_fnc fnc, void * fnc_handle)

are these timings valid?

Parameters

const struct v4l2_dv_timings * t the v4l2_dv_timings struct.

const struct v4l2_dv_timings_cap * cap the v4l2_dv_timings_cap capabilities.

v4l2_check_dv_timings_fnc fnc callback to check if this timing is OK. May be NULL.

void * fnc_handle a handle that is passed on to **fnc**.

Description

Returns true if the given dv_timings struct is supported by the hardware capabilities and the callback function (if non-NULL), returns false otherwise.

int **v4l2_enum_dv_timings_cap**(struct v4l2_enum_dv_timings * t, const struct v4l2_dv_timings_cap * cap, v4l2_check_dv_timings_fnc fnc, void * fnc_handle)

Helper function to enumerate possible DV timings based on capabilities

Parameters

struct v4l2_enum_dv_timings * t the v4l2_enum_dv_timings struct.

const struct v4l2_dv_timings_cap * cap the v4l2_dv_timings_cap capabilities.

v4l2_check_dv_timings_fnc fnc callback to check if this timing is OK. May be NULL.

void * fnc_handle a handle that is passed on to **fnc**.

Description

This enumerates dv_timings using the full list of possible CEA-861 and DMT timings, filtering out any timings that are not supported based on the hardware capabilities and the callback function (if non-NULL).

If a valid timing for the given index is found, it will fill in **t** and return 0, otherwise it returns -EINVAL.

bool **v4l2_find_dv_timings_cap**(struct v4l2_dv_timings * t, const struct v4l2_dv_timings_cap * cap, unsigned pclock_delta, v4l2_check_dv_timings_fnc fnc, void * fnc_handle)

Find the closest timings struct

Parameters

struct v4l2_dv_timings * t the v4l2_enum_dv_timings struct.

const struct v4l2_dv_timings_cap * cap the v4l2_dv_timings_cap capabilities.

unsigned pclock_delta maximum delta between t->pixelclock and the timing struct under consideration.

v4l2_check_dv_timings_fnc fnc callback to check if a given timings struct is OK. May be NULL.

void * fnc_handle a handle that is passed on to **fnc**.

Description

This function tries to map the given timings to an entry in the full list of possible CEA-861 and DMT timings, filtering out any timings that are not supported based on the hardware capabilities and the callback function (if non-NULL).

On success it will fill in **t** with the found timings and it returns true. On failure it will return false.

bool **v4l2_find_dv_timings_cea861_vic**(struct v4l2_dv_timings * t, u8 vic)
find timings based on CEA-861 VIC

Parameters

struct v4l2_dv_timings * t the timings data.

u8 vic CEA-861 VIC code

Description

On success it will fill in **t** with the found timings and it returns true. On failure it will return false.

bool **v4l2_match_dv_timings**(const struct v4l2_dv_timings * measured, const struct v4l2_dv_timings * standard, unsigned pclock_delta, bool match_reduced_fps)
do two timings match?

Parameters

const struct v4l2_dv_timings * measured the measured timings data.

const struct v4l2_dv_timings * standard the timings according to the standard.

unsigned pclock_delta maximum delta in Hz between standard->pixelclock and the measured timings.

bool match_reduced_fps if true, then fail if V4L2_DV_FL_REDUCED_FPS does not match.

Description

Returns true if the two timings match, returns false otherwise.

void **v4l2_print_dv_timings**(const char * dev_prefix, const char * prefix, const struct v4l2_dv_timings * t, bool detailed)
log the contents of a dv_timings struct

Parameters

const char * dev_prefix device prefix for each log line.

const char * prefix additional prefix for each log line, may be NULL.

const struct v4l2_dv_timings * t the timings data.

bool detailed if true, give a detailed log.

bool **v4l2_detect_cvt**(unsigned frame_height, unsigned hfreq, unsigned vsync, unsigned active_width, u32 polarities, bool interlaced, struct v4l2_dv_timings * fmt)

detect if the given timings follow the CVT standard

Parameters

unsigned frame_height the total height of the frame (including blanking) in lines.

unsigned hfreq the horizontal frequency in Hz.

unsigned vsync the height of the vertical sync in lines.

unsigned active_width active width of image (does not include blanking). This information is needed only in case of version 2 of reduced blanking. In other cases, this parameter does not have any effect on timings.

u32 polarities the horizontal and vertical polarities (same as struct v4l2_bt_timings polarities).

bool interlaced if this flag is true, it indicates interlaced format

struct v4l2_dv_timings * fmt the resulting timings.

Description

This function will attempt to detect if the given values correspond to a valid CVT format. If so, then it will return true, and fmt will be filled in with the found CVT timings.

bool **v4l2_detect_gtf**(unsigned frame_height, unsigned hfreq, unsigned vsync, u32 polarities, bool interlaced, struct v4l2_fract aspect, struct v4l2_dv_timings * fmt)

detect if the given timings follow the GTF standard

Parameters

unsigned frame_height the total height of the frame (including blanking) in lines.

unsigned hfreq the horizontal frequency in Hz.

unsigned vsync the height of the vertical sync in lines.

u32 polarities the horizontal and vertical polarities (same as struct v4l2_bt_timings polarities).

bool interlaced if this flag is true, it indicates interlaced format

struct v4l2_fract aspect preferred aspect ratio. GTF has no method of determining the aspect ratio in order to derive the image width from the image height, so it has to be passed explicitly. Usually the native screen aspect ratio is used for this. If it is not filled in correctly, then 16:9 will be assumed.

struct v4l2_dv_timings * fmt the resulting timings.

Description

This function will attempt to detect if the given values correspond to a valid GTF format. If so, then it will return true, and fmt will be filled in with the found GTF timings.

struct v4l2_fract **v4l2_calc_aspect_ratio**(u8 hor_landscape,
u8 vert_portrait)
calculate the aspect ratio based on bytes 0x15 and 0x16 from the EDID.

Parameters

u8 hor_landscape byte 0x15 from the EDID.

u8 vert_portrait byte 0x16 from the EDID.

Description

Determines the aspect ratio from the EDID. See VESA Enhanced EDID standard, release A, rev 2, section 3.6.2: "Horizontal and Vertical Screen Size or Aspect Ratio"

struct v4l2_fract **v4l2_dv_timings_aspect_ratio**(const struct v4l2_dv_timings * t)
calculate the aspect ratio based on the v4l2_dv_timings information.

Parameters

const struct v4l2_dv_timings * t the timings data.

bool **can_reduce_fps**(struct v4l2_bt_timings * bt)
check if conditions for reduced fps are true.

Parameters

struct v4l2_bt_timings * bt v4l2 timing structure

Description

For different timings reduced fps is allowed if the following conditions are met:

- For CVT timings: if reduced blanking v2 (vsync == 8) is true.
- For CEA861 timings: if `V4L2_DV_FL_CAN_REDUCE_FPS` flag is true.

struct **v4l2_hdmi_colorimetry**
describes the HDMI colorimetry information

Definition

```
struct v4l2_hdmi_colorimetry {
  enum v4l2_colorspace colorspace;
  enum v4l2_ycbcr_encoding ycbcr_enc;
  enum v4l2_quantization quantization;
  enum v4l2_xfer_func xfer_func;
};
```

Members

colorspace enum v4l2_colorspace, the colorspace

ycbcr_enc enum v4l2_ycbcr_encoding, Y' CbCr encoding

quantization enum v4l2_quantization, colorspace quantization

xfer_func enum v4l2_xfer_func, colorspace transfer function

53.1.18 V4L2 flash functions and data structures

struct **v4l2_flash_ctrl_data**
flash control initialization data, filled basing on the features declared by the LED flash class driver in the v4l2_flash_config

Definition

```
struct v4l2_flash_ctrl_data {
  struct v4l2_ctrl_config config;
  u32 cid;
};
```

Members

config initialization data for a control

cid contains v4l2 flash control id if the config field was initialized, 0 otherwise

struct **v4l2_flash_ops**
V4L2 flash operations

Definition

```
struct v4l2_flash_ops {
  int (*external_strobe_set)(struct v4l2_flash *v4l2_flash, bool enable);
  enum led_brightness (*intensity_to_led_brightness) (struct v4l2_flash␣
↪*v4l2_flash, s32 intensity);
  s32 (*led_brightness_to_intensity) (struct v4l2_flash *v4l2_flash, enum␣
↪led_brightness);
};
```

Members

external_strobe_set Setup strobing the flash by hardware pin state assertion.

intensity_to_led_brightness Convert intensity to brightness in a device specific manner

led_brightness_to_intensity convert brightness to intensity in a device specific manner.

struct **v4l2_flash_config**
V4L2 Flash sub-device initialization data

Definition

```
struct v4l2_flash_config {
  char dev_name[32];
  struct led_flash_setting intensity;
  u32 flash_faults;
  unsigned int has_external_strobe:1;
};
```

Members

dev_name the name of the media entity, unique in the system

intensity non-flash strobe constraints for the LED

flash_faults bitmask of flash faults that the LED flash class device can report; corresponding LED_FAULT* bit definitions are available in the header file <linux/led-class-flash.h>

has_external_strobe external strobe capability

struct **v4l2_flash**
Flash sub-device context

Definition

```
struct v4l2_flash {
  struct led_classdev_flash *fled_cdev;
  struct led_classdev *iled_cdev;
  const struct v4l2_flash_ops *ops;
  struct v4l2_subdev sd;
  struct v4l2_ctrl_handler hdl;
  struct v4l2_ctrl **ctrls;
};
```

Members

fled_cdev LED flash class device controlled by this sub-device

iled_cdev LED class device representing indicator LED associated with the LED flash class device

ops V4L2 specific flash ops

sd V4L2 sub-device

hdl flash controls handler

ctrls array of pointers to controls, whose values define the sub-device state

struct v4l2_flash * **v4l2_subdev_to_v4l2_flash**(struct v4l2_subdev * sd)
Returns a `struct v4l2_flash` from the `struct v4l2_subdev` embedded on it.

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

struct v4l2_flash * **v4l2_ctrl_to_v4l2_flash**(struct v4l2_ctrl * c)
Returns a `struct v4l2_flash` from the `struct v4l2_ctrl` embedded on it.

Parameters

struct v4l2_ctrl * c pointer to `struct v4l2_ctrl`

struct v4l2_flash * **v4l2_flash_init**(struct device * dev, struct fwnode_handle * fwn, struct led_classdev_flash * fled_cdev, const struct v4l2_flash_ops * ops, struct v4l2_flash_config * config)
initialize V4L2 flash led sub-device

Parameters

struct device * dev flash device, e.g. an I2C device

struct fwnode_handle * fwn fwnode_handle of the LED, may be NULL if the same as device's

struct led_classdev_flash * fled_cdev LED flash class device to wrap

const struct v4l2_flash_ops * ops V4L2 Flash device ops

struct v4l2_flash_config * config initialization data for V4L2 Flash sub-device

Description

Create V4L2 Flash sub-device wrapping given LED subsystem device. The ops pointer is stored by the V4L2 flash framework. No references are held to config nor its contents once this function has returned.

Return

A valid pointer, or, when an error occurs, the return value is encoded using ERR_PTR(). Use IS_ERR() to check and PTR_ERR() to obtain the numeric return value.

struct v4l2_flash * **v4l2_flash_indicator_init**(struct device * dev, struct fwnode_handle * fwn, struct led_classdev * iled_cdev, struct v4l2_flash_config * config)
initialize V4L2 indicator sub-device

Parameters

struct device * dev flash device, e.g. an I2C device

struct fwnode_handle * fwn fwnode_handle of the LED, may be NULL if the same as device's

struct led_classdev * iled_cdev LED flash class device representing the indicator LED

struct v4l2_flash_config * config initialization data for V4L2 Flash sub-device

Description

Create V4L2 Flash sub-device wrapping given LED subsystem device. The ops pointer is stored by the V4L2 flash framework. No references are held to config nor its contents once this function has returned.

Return

A valid pointer, or, when an error occurs, the return value is encoded using ERR_PTR(). Use IS_ERR() to check and PTR_ERR() to obtain the numeric return value.

void **v4l2_flash_release**(struct v4l2_flash * v4l2_flash)
release V4L2 Flash sub-device

Parameters

struct v4l2_flash * v4l2_flash the V4L2 Flash sub-device to release

Description

Release V4L2 Flash sub-device.

53.1.19 V4L2 Media Controller functions and data structures

int **v4l2_mc_create_media_graph**(struct media_device * mdev)
create Media Controller links at the graph.

Parameters

struct media_device * mdev pointer to the media_device struct.

Description

Add links between the entities commonly found on PC customer's hardware at the V4L2 side: camera sensors, audio and video PLL-IF decoders, tuners, analog TV decoder and I/O entities (video, VBI and Software Defined Radio).

Note: Webcams are modelled on a very simple way: the sensor is connected directly to the I/O entity. All dirty details, like scaler and crop HW are hidden. While such mapping is enough for v4l2 interface centric PC-consumer's hardware, V4L2 subdev centric camera hardware should not use this routine, as it will not build the right graph.

---

int **v4l_enable_media_source**(struct video_device * vdev)
Hold media source for exclusive use if free

Parameters

struct video_device * vdev pointer to struct video_device

Description

This interface calls enable_source handler to determine if media source is free for use. The enable_source handler is responsible for checking is the media source is free and start a pipeline between the media source and the media entity associated with the video device. This interface should be called from v4l2-core and dvb-core interfaces that change the source configuration.

Return

returns zero on success or a negative error code.

void **v4l_disable_media_source**(struct video_device * vdev)
Release media source

Parameters

struct video_device * vdev pointer to struct video_device

Description

This interface calls disable_source handler to release the media source. The disable_source handler stops the active media pipeline between the media source and the media entity associated with the video device.

Return

returns zero on success or a negative error code.

int **v4l2_create_fwnode_links_to_pad**(struct v4l2_subdev * src_sd, struct media_pad * sink)
Create fwnode-based links from a source subdev to a sink subdev pad.

Parameters

struct v4l2_subdev * src_sd undescribed

struct media_pad * sink undescribed

Description

src_sd - pointer to a source subdev **sink** - pointer to a subdev sink pad

This function searches for fwnode endpoint connections from a source subdevice to a single sink pad, and if suitable connections are found, translates them into media links to that pad. The function can be called by the sink subdevice, in its v4l2-async notifier subdev bound callback, to create links from a bound source subdevice.

Note: Any sink subdevice that calls this function must implement the .get_fwnode_pad media operation in order to verify endpoints passed to the sink are owned by the sink.

Return 0 on success or a negative error code on failure.

int **v4l2_create_fwnode_links**(struct v4l2_subdev * src_sd, struct v4l2_subdev * sink_sd)
Create fwnode-based links from a source subdev to a sink subdev.

Parameters

struct v4l2_subdev * src_sd undescribed

struct v4l2_subdev * sink_sd undescribed

Description

src_sd - pointer to a source subdevice **sink_sd** - pointer to a sink subdevice

This function searches for any and all fwnode endpoint connections between source and sink subdevices, and translates them into media links. The function can be called by the sink subdevice, in its v4l2-async notifier subdev bound callback, to create all links from a bound source subdevice.

Note: Any sink subdevice that calls this function must implement the .get_fwnode_pad media operation in order to verify endpoints passed to the sink are owned by the sink.

Return 0 on success or a negative error code on failure.

int **v4l2_pipeline_pm_get**(struct media_entity * entity)
Increase the use count of a pipeline

Parameters

struct media_entity * entity The root entity of a pipeline

Description

Update the use count of all entities in the pipeline and power entities on.

This function is intended to be called in video node open. It uses struct media_entity.use_count to track the power status. The use of this function should be paired with `v4l2_pipeline_link_notify()`.

Return 0 on success or a negative error code on failure.

void **v4l2_pipeline_pm_put**(struct media_entity * entity)
Decrease the use count of a pipeline

Parameters

struct media_entity * entity The root entity of a pipeline

Description

Update the use count of all entities in the pipeline and power entities off.

This function is intended to be called in video node release. It uses struct media_entity.use_count to track the power status. The use of this function should be paired with `v4l2_pipeline_link_notify()`.

int **v4l2_pipeline_link_notify**(struct media_link * link, u32 flags, unsigned int notification)
Link management notification callback

Parameters

struct media_link * link The link

u32 flags New link flags that will be applied

unsigned int notification The link's state change notification type (MEDIA_DEV_NOTIFY_*)

Description

React to link management on powered pipelines by updating the use count of all entities in the source and sink sides of the link. Entities are powered on or off accordingly. The use of this function should be paired with v4l2_pipeline_pm_{get,put}().

Return 0 on success or a negative error code on failure. Powering entities off is assumed to never fail. This function will not fail for disconnection events.

53.1.20 V4L2 Media Bus functions and data structures

enum **v4l2_mbus_type**
media bus type

Constants

V4L2_MBUS_UNKNOWN unknown bus type, no V4L2 mediabus configuration

V4L2_MBUS_PARALLEL parallel interface with hsync and vsync

V4L2_MBUS_BT656 parallel interface with embedded synchronisation, can also be used for BT.1120

V4L2_MBUS_CSI1 MIPI CSI-1 serial interface

V4L2_MBUS_CCP2 CCP2 (Compact Camera Port 2)

V4L2_MBUS_CSI2_DPHY MIPI CSI-2 serial interface, with D-PHY

V4L2_MBUS_CSI2_CPHY MIPI CSI-2 serial interface, with C-PHY

struct **v4l2_mbus_config**
media bus configuration

Definition

```
struct v4l2_mbus_config {
  enum v4l2_mbus_type type;
  unsigned int flags;
};
```

Members

type in: interface type

flags in / out: configuration flags, depending on **type**

void **v4l2_fill_pix_format**(struct v4l2_pix_format * pix_fmt, const struct v4l2_mbus_framefmt * mbus_fmt)
Ancillary routine that fills a `struct v4l2_pix_format` fields from a `struct v4l2_mbus_framefmt`.

Parameters

struct v4l2_pix_format * pix_fmt pointer to `struct v4l2_pix_format` to be filled

const struct v4l2_mbus_framefmt * mbus_fmt pointer to `struct v4l2_mbus_framefmt` to be used as model

void **v4l2_fill_mbus_format**(struct v4l2_mbus_framefmt * mbus_fmt, const struct v4l2_pix_format * pix_fmt, u32 code)
Ancillary routine that fills a `struct v4l2_mbus_framefmt` from a `struct v4l2_pix_format` and a data format code.

Parameters

struct v4l2_mbus_framefmt * mbus_fmt pointer to `struct v4l2_mbus_framefmt` to be filled

const struct v4l2_pix_format * pix_fmt pointer to `struct v4l2_pix_format` to be used as model

u32 code data format code (from `enum v4l2_mbus_pixelcode`)

void **v4l2_fill_pix_format_mplane**(struct v4l2_pix_format_mplane * pix_mp_fmt, const struct v4l2_mbus_framefmt * mbus_fmt)
Ancillary routine that fills a `struct v4l2_pix_format_mplane` fields from a media bus structure.

Parameters

struct v4l2_pix_format_mplane * pix_mp_fmt pointer to `struct v4l2_pix_format_mplane` to be filled

const struct v4l2_mbus_framefmt * mbus_fmt pointer to struct v4l2_mbus_framefmt to be used as model

void **v4l2_fill_mbus_format_mplane**(struct v4l2_mbus_framefmt * mbus_fmt, const struct v4l2_pix_format_mplane * pix_mp_fmt)

Ancillary routine that fills a struct v4l2_mbus_framefmt from a struct v4l2_pix_format_mplane.

Parameters

struct v4l2_mbus_framefmt * mbus_fmt pointer to struct v4l2_mbus_framefmt to be filled

const struct v4l2_pix_format_mplane * pix_mp_fmt pointer to struct v4l2_pix_format_mplane to be used as model

53.1.21 V4L2 Memory to Memory functions and data structures

struct **v4l2_m2m_ops**

mem-to-mem device driver callbacks

Definition

```
struct v4l2_m2m_ops {
  void (*device_run)(void *priv);
  int (*job_ready)(void *priv);
  void (*job_abort)(void *priv);
};
```

Members

device_run required. Begin the actual job (transaction) inside this callback. The job does NOT have to end before this callback returns (and it will be the usual case). When the job finishes, v4l2_m2m_job_finish() or v4l2_m2m_buf_done_and_job_finish() has to be called.

job_ready optional. Should return 0 if the driver does not have a job fully prepared to run yet (i.e. it will not be able to finish a transaction without sleeping). If not provided, it will be assumed that one source and one destination buffer are all that is required for the driver to perform one full transaction. This method may not sleep.

job_abort optional. Informs the driver that it has to abort the currently running transaction as soon as possible (i.e. as soon as it can stop the device safely; e.g. in the next interrupt handler), even if the transaction would not have been finished by then. After the driver performs the necessary steps, it has to call v4l2_m2m_job_finish() or v4l2_m2m_buf_done_and_job_finish() as if the transaction ended normally. This function does not have to (and will usually not) wait until the device enters a state when it can be stopped.

struct **v4l2_m2m_queue_ctx**

represents a queue for buffers ready to be processed

Definition

```
struct v4l2_m2m_queue_ctx {
  struct vb2_queue        q;
  struct list_head        rdy_queue;
  spinlock_t rdy_spinlock;
  u8 num_rdy;
  bool buffered;
};
```

Members

q pointer to struct vb2_queue

rdy_queue List of V4L2 mem-to-mem queues

rdy_spinlock spin lock to protect the struct usage

num_rdy number of buffers ready to be processed

buffered is the queue buffered?

Description

Queue for buffers ready to be processed as soon as this instance receives access to the device.

struct **v4l2_m2m_ctx**
    Memory to memory context structure

Definition

```
struct v4l2_m2m_ctx {
  struct mutex                    *q_lock;
  bool new_frame;
  bool is_draining;
  struct vb2_v4l2_buffer          *last_src_buf;
  bool next_buf_last;
  bool has_stopped;
  struct v4l2_m2m_dev             *m2m_dev;
  struct v4l2_m2m_queue_ctx       cap_q_ctx;
  struct v4l2_m2m_queue_ctx       out_q_ctx;
  struct list_head                queue;
  unsigned long                   job_flags;
  wait_queue_head_t finished;
  void *priv;
};
```

Members

q_lock struct mutex lock

new_frame valid in the device_run callback: if true, then this starts a new frame; if false, then this is a new slice for an existing frame. This is always true unless V4L2_BUF_CAP_SUPPORTS_M2M_HOLD_CAPTURE_BUF is set, which indicates slicing support.

is_draining indicates device is in draining phase

last_src_buf indicate the last source buffer for draining

next_buf_last next capture queud buffer will be tagged as last

has_stopped indicate the device has been stopped

m2m_dev opaque pointer to the internal data to handle M2M context

cap_q_ctx Capture (output to memory) queue context

out_q_ctx Output (input from memory) queue context

queue List of memory to memory contexts

job_flags Job queue flags, used internally by v4l2-mem2mem.c: `TRANS_QUEUED`, `TRANS_RUNNING` and `TRANS_ABORT`.

finished Wait queue used to signalize when a job queue finished.

priv Instance private data

Description

The memory to memory context is specific to a file handle, NOT to e.g. a device.

struct **v4l2_m2m_buffer**
Memory to memory buffer

Definition

```
struct v4l2_m2m_buffer {
  struct vb2_v4l2_buffer  vb;
  struct list_head        list;
};
```

Members

vb pointer to struct `vb2_v4l2_buffer`

list list of m2m buffers

void * **v4l2_m2m_get_curr_priv**(struct v4l2_m2m_dev * m2m_dev)
return driver private data for the currently running instance or NULL if no instance is running

Parameters

struct v4l2_m2m_dev * m2m_dev opaque pointer to the internal data to handle M2M context

struct vb2_queue * **v4l2_m2m_get_vq**(struct v4l2_m2m_ctx * m2m_ctx,
enum v4l2_buf_type type)
return vb2_queue for the given type

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

enum v4l2_buf_type type type of the V4L2 buffer, as defined by enum `v4l2_buf_type`

void **v4l2_m2m_try_schedule**(struct v4l2_m2m_ctx * m2m_ctx)
check whether an instance is ready to be added to the pending job queue and add it if so.

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

Description

There are three basic requirements an instance has to meet to be able to run: 1) at least one source buffer has to be queued, 2) at least one destination buffer has to be queued, 3) streaming has to be on.

If a queue is buffered (for example a decoder hardware ringbuffer that has to be drained before doing streamoff), allow scheduling without v4l2 buffers on that queue.

There may also be additional, custom requirements. In such case the driver should supply a custom callback (job_ready in v4l2_m2m_ops) that should return 1 if the instance is ready. An example of the above could be an instance that requires more than one src/dst buffer per transaction.

void **v4l2_m2m_job_finish**(struct v4l2_m2m_dev * m2m_dev, struct v4l2_m2m_ctx * m2m_ctx)

inform the framework that a job has been finished and have it clean up

Parameters

struct v4l2_m2m_dev * m2m_dev opaque pointer to the internal data to handle M2M context

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

Description

Called by a driver to yield back the device after it has finished with it. Should be called as soon as possible after reaching a state which allows other instances to take control of the device.

This function has to be called only after `v4l2_m2m_ops->device_run` callback has been called on the driver. To prevent recursion, it should not be called directly from the `v4l2_m2m_ops->device_run` callback though.

void **v4l2_m2m_buf_done_and_job_finish**(struct v4l2_m2m_dev * m2m_dev, struct v4l2_m2m_ctx * m2m_ctx, enum vb2_buffer_state state)

return source/destination buffers with state and inform the framework that a job has been finished and have it clean up

Parameters

struct v4l2_m2m_dev * m2m_dev opaque pointer to the internal data to handle M2M context

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

enum vb2_buffer_state state vb2 buffer state passed to v4l2_m2m_buf_done().

Description

Drivers that set V4L2_BUF_CAP_SUPPORTS_M2M_HOLD_CAPTURE_BUF must use this function instead of job_finish() to take held buffers into account. It is optional for other drivers.

This function removes the source buffer from the ready list and returns it with the given state. The same is done for the destination buffer, unless it is marked 'held'. In that case the buffer is kept on the ready list.

After that the job is finished (see job_finish()).

This allows for multiple output buffers to be used to fill in a single capture buffer. This is typically used by stateless decoders where multiple e.g. H.264 slices contribute to a single decoded frame.

void **v4l2_m2m_clear_state**(struct v4l2_m2m_ctx * m2m_ctx)
clear encoding/decoding state

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

void **v4l2_m2m_mark_stopped**(struct v4l2_m2m_ctx * m2m_ctx)
set current encoding/decoding state as stopped

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

bool **v4l2_m2m_dst_buf_is_last**(struct v4l2_m2m_ctx * m2m_ctx)
return the current encoding/decoding session draining management state of next queued capture buffer

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

Description

This last capture buffer should be tagged with V4L2_BUF_FLAG_LAST to notify the end of the capture session.

bool **v4l2_m2m_has_stopped**(struct v4l2_m2m_ctx * m2m_ctx)
return the current encoding/decoding session stopped state

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

bool **v4l2_m2m_is_last_draining_src_buf**(struct v4l2_m2m_ctx * m2m_ctx, struct vb2_v4l2_buffer * vbuf)
return the output buffer draining state in the current encoding/decoding session

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_v4l2_buffer * vbuf pointer to struct v4l2_buffer

Description

This will identify the last output buffer queued before a session stop was required, leading to an actual encoding/decoding session stop state in the encoding/decoding process after being processed.

void **v4l2_m2m_last_buffer_done**(struct v4l2_m2m_ctx * m2m_ctx, struct vb2_v4l2_buffer * vbuf)
marks the buffer with LAST flag and DONE

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_v4l2_buffer * vbuf pointer to struct v4l2_buffer

int **v4l2_m2m_reqbufs**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_requestbuffers * reqbufs)
multi-queue-aware REQBUFS multiplexer

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_requestbuffers * reqbufs pointer to struct v4l2_requestbuffers

int **v4l2_m2m_querybuf**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_buffer * buf)
multi-queue-aware QUERYBUF multiplexer

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_buffer * buf pointer to struct v4l2_buffer

Description

See v4l2_m2m_mmap() documentation for details.

int **v4l2_m2m_qbuf**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_buffer * buf)
enqueue a source or destination buffer, depending on the type

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_buffer * buf pointer to struct v4l2_buffer

int **v4l2_m2m_dqbuf**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_buffer * buf)
dequeue a source or destination buffer, depending on the type

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_buffer * buf pointer to struct v4l2_buffer

int **v4l2_m2m_prepare_buf**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_buffer * buf)
prepare a source or destination buffer, depending on the type

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_buffer * buf pointer to struct v4l2_buffer

int **v4l2_m2m_create_bufs**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_create_buffers * create)
create a source or destination buffer, depending on the type

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_create_buffers * create pointer to struct v4l2_create_buffers

int **v4l2_m2m_expbuf**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_exportbuffer * eb)
export a source or destination buffer, depending on the type

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_exportbuffer * eb pointer to struct v4l2_exportbuffer

int **v4l2_m2m_streamon**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, enum v4l2_buf_type type)
turn on streaming for a video queue

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

enum v4l2_buf_type type type of the V4L2 buffer, as defined by enum v4l2_buf_type

int **v4l2_m2m_streamoff**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, enum v4l2_buf_type type)
turn off streaming for a video queue

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

enum v4l2_buf_type type type of the V4L2 buffer, as defined by enum v4l2_buf_type

void **v4l2_m2m_update_start_streaming_state**(struct v4l2_m2m_ctx * m2m_ctx, struct vb2_queue * q)
update the encoding/decoding session state when a start of streaming of a video queue is requested

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_queue * q queue

void **v4l2_m2m_update_stop_streaming_state**(struct v4l2_m2m_ctx * m2m_ctx, struct vb2_queue * q)
update the encoding/decoding session state when a stop of streaming of a video queue is requested

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_queue * q queue

int **v4l2_m2m_encoder_cmd**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_encoder_cmd * ec)
execute an encoder command

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_encoder_cmd * ec pointer to the encoder command

int **v4l2_m2m_decoder_cmd**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct v4l2_decoder_cmd * dc)
execute a decoder command

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct v4l2_decoder_cmd * dc pointer to the decoder command

__poll_t **v4l2_m2m_poll**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct poll_table_struct * wait)
poll replacement, for destination buffers only

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct poll_table_struct * wait pointer to struct poll_table_struct

Description

Call from the driver's poll() function. Will poll both queues. If a buffer is available to dequeue (with dqbuf) from the source queue, this will indicate that a non-blocking write can be performed, while read will be returned in case of the destination queue.

int **v4l2_m2m_mmap**(struct file * file, struct v4l2_m2m_ctx * m2m_ctx, struct vm_area_struct * vma)
source and destination queues-aware mmap multiplexer

Parameters

struct file * file pointer to struct file

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vm_area_struct * vma pointer to struct vm_area_struct

Description

Call from driver's mmap() function. Will handle mmap() for both queues seamlessly for videobuffer, which will receive normal per-queue offsets and proper videobuf queue pointers. The differentiation is made outside videobuf by adding a predefined offset to buffers from one of the queues and subtracting it before passing it back to videobuf. Only drivers (and thus applications) receive modified offsets.

struct v4l2_m2m_dev * **v4l2_m2m_init**(const struct v4l2_m2m_ops * m2m_ops)
initialize per-driver m2m data

Parameters

const struct v4l2_m2m_ops * m2m_ops pointer to struct v4l2_m2m_ops

Description

Usually called from driver's probe() function.

Return

returns an opaque pointer to the internal data to handle M2M context

void **v4l2_m2m_release**(struct v4l2_m2m_dev * m2m_dev)
cleans up and frees a m2m_dev structure

Parameters

struct v4l2_m2m_dev * m2m_dev opaque pointer to the internal data to handle M2M context

Description

Usually called from driver's `remove()` function.

struct v4l2_m2m_ctx * **v4l2_m2m_ctx_init**(struct v4l2_m2m_dev * m2m_dev, void * drv_priv, int (*queue_init)(void *priv, struct vb2_queue *src_vq, struct vb2_queue *dst_vq))
allocate and initialize a m2m context

Parameters

struct v4l2_m2m_dev * m2m_dev opaque pointer to the internal data to handle M2M context

void * drv_priv driver's instance private data

int (*)(void *priv, struct vb2_queue *src_vq, struct vb2_queue *dst_vq) queue_in
a callback for queue type-specific initialization function to be used for initializing videobuf_queues

Description

Usually called from driver's `open()` function.

void **v4l2_m2m_ctx_release**(struct v4l2_m2m_ctx * m2m_ctx)
release m2m context

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

Description

Usually called from driver's release() function.

void **v4l2_m2m_buf_queue**(struct v4l2_m2m_ctx * m2m_ctx, struct vb2_v4l2_buffer * vbuf)
add a buffer to the proper ready buffers list.

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

struct vb2_v4l2_buffer * vbuf pointer to struct `vb2_v4l2_buffer`

Description

Call from videobuf_queue_ops->ops->buf_queue, videobuf_queue_ops callback.

unsigned int **v4l2_m2m_num_src_bufs_ready**(struct v4l2_m2m_ctx * m2m_ctx)
return the number of source buffers ready for use

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

unsigned int **v4l2_m2m_num_dst_bufs_ready**(struct v4l2_m2m_ctx * m2m_ctx)
return the number of destination buffers ready for use

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_v4l2_buffer * **v4l2_m2m_next_buf**(struct v4l2_m2m_queue_ctx * q_ctx)
return next buffer from the list of ready buffers

Parameters

struct v4l2_m2m_queue_ctx * q_ctx pointer to struct **v4l2_m2m_queue_ctx**

struct vb2_v4l2_buffer * **v4l2_m2m_next_src_buf**(struct v4l2_m2m_ctx * m2m_ctx)
return next source buffer from the list of ready buffers

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_v4l2_buffer * **v4l2_m2m_next_dst_buf**(struct v4l2_m2m_ctx * m2m_ctx)
return next destination buffer from the list of ready buffers

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_v4l2_buffer * **v4l2_m2m_last_buf**(struct v4l2_m2m_queue_ctx * q_ctx)
return last buffer from the list of ready buffers

Parameters

struct v4l2_m2m_queue_ctx * q_ctx pointer to struct **v4l2_m2m_queue_ctx**

struct vb2_v4l2_buffer * **v4l2_m2m_last_src_buf**(struct v4l2_m2m_ctx * m2m_ctx)
return last destination buffer from the list of ready buffers

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_v4l2_buffer * **v4l2_m2m_last_dst_buf**(struct v4l2_m2m_ctx * m2m_ctx)
return last destination buffer from the list of ready buffers

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

v4l2_m2m_for_each_dst_buf(m2m_ctx, b)
iterate over a list of destination ready buffers

Parameters

m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

b current buffer of type struct v4l2_m2m_buffer

v4l2_m2m_for_each_src_buf(m2m_ctx, b)
iterate over a list of source ready buffers

Parameters

m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

b current buffer of type struct v4l2_m2m_buffer

v4l2_m2m_for_each_dst_buf_safe(m2m_ctx, b, n)
iterate over a list of destination ready buffers safely

Parameters

m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

b current buffer of type struct v4l2_m2m_buffer

n used as temporary storage

v4l2_m2m_for_each_src_buf_safe(m2m_ctx, b, n)
iterate over a list of source ready buffers safely

Parameters

m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

b current buffer of type struct v4l2_m2m_buffer

n used as temporary storage

struct vb2_queue * **v4l2_m2m_get_src_vq**(struct v4l2_m2m_ctx * m2m_ctx)
return vb2_queue for source buffers

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_queue * **v4l2_m2m_get_dst_vq**(struct v4l2_m2m_ctx * m2m_ctx)
return vb2_queue for destination buffers

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct v4l2_m2m_ctx

struct vb2_v4l2_buffer * **v4l2_m2m_buf_remove**(struct v4l2_m2m_queue_ctx * q_ctx)
take off a buffer from the list of ready buffers and return it

Parameters

struct v4l2_m2m_queue_ctx * q_ctx pointer to struct **v4l2_m2m_queue_ctx**

struct vb2_v4l2_buffer * **v4l2_m2m_src_buf_remove**(struct v4l2_m2m_ctx * m2m_ctx)
take off a source buffer from the list of ready buffers and return it

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

struct vb2_v4l2_buffer * **v4l2_m2m_dst_buf_remove**(struct v4l2_m2m_ctx * m2m_ctx)
take off a destination buffer from the list of ready buffers and return it

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

void **v4l2_m2m_buf_remove_by_buf**(struct v4l2_m2m_queue_ctx * q_ctx, struct vb2_v4l2_buffer * vbuf)
take off exact buffer from the list of ready buffers

Parameters

struct v4l2_m2m_queue_ctx * q_ctx pointer to struct **v4l2_m2m_queue_ctx**

struct vb2_v4l2_buffer * vbuf the buffer to be removed

void **v4l2_m2m_src_buf_remove_by_buf**(struct v4l2_m2m_ctx * m2m_ctx, struct vb2_v4l2_buffer * vbuf)
take off exact source buffer from the list of ready buffers

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

struct vb2_v4l2_buffer * vbuf the buffer to be removed

void **v4l2_m2m_dst_buf_remove_by_buf**(struct v4l2_m2m_ctx * m2m_ctx, struct vb2_v4l2_buffer * vbuf)
take off exact destination buffer from the list of ready buffers

Parameters

struct v4l2_m2m_ctx * m2m_ctx m2m context assigned to the instance given by struct `v4l2_m2m_ctx`

struct vb2_v4l2_buffer * vbuf the buffer to be removed

void **v4l2_m2m_buf_copy_metadata**(const struct vb2_v4l2_buffer * out_vb, struct vb2_v4l2_buffer * cap_vb, bool copy_frame_flags)
copy buffer metadata from the output buffer to the capture buffer

Parameters

const struct vb2_v4l2_buffer * out_vb the output buffer that is the source of the metadata.

struct vb2_v4l2_buffer * cap_vb the capture buffer that will receive the metadata.

bool copy_frame_flags copy the KEY/B/PFRAME flags as well.

Description

This helper function copies the timestamp, timecode (if the TIMECODE buffer flag was set), field and the TIMECODE, KEYFRAME, BFRAME, PFRAME and TSTAMP_SRC_MASK flags from **out_vb** to **cap_vb**.

If **copy_frame_flags** is false, then the KEYFRAME, BFRAME and PFRAME flags are not copied. This is typically needed for encoders that set this bits explicitly.

53.1.22 V4L2 async kAPI

enum **v4l2_async_match_type**
type of asynchronous subdevice logic to be used in order to identify a match

Constants

V4L2_ASYNC_MATCH_CUSTOM Match will use the logic provided by `struct v4l2_async_subdev.match` ops

V4L2_ASYNC_MATCH_DEVNAME Match will use the device name

V4L2_ASYNC_MATCH_I2C Match will check for I2C adapter ID and address

V4L2_ASYNC_MATCH_FWNODE Match will use firmware node

Description

This enum is used by the asyncrhronous sub-device logic to define the algorithm that will be used to match an asynchronous device.

struct **v4l2_async_subdev**
sub-device descriptor, as known to a bridge

Definition

```
struct v4l2_async_subdev {
  enum v4l2_async_match_type match_type;
  union {
    struct fwnode_handle *fwnode;
    const char *device_name;
    struct {
      int adapter_id;
      unsigned short address;
    } i2c;
    struct {
      bool (*match)(struct device *dev, struct v4l2_async_subdev *sd);
      void *priv;
    } custom;
  } match;
```

(continues on next page)

(continued from previous page)

```
  struct list_head list;
  struct list_head asd_list;
};
```

Members

match_type type of match that will be used

match union of per-bus type matching data sets

match.fwnode pointer to `struct fwnode_handle` to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_FWNODE`.

match.device_name string containing the device name to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_DEVNAME`.

match.i2c embedded struct with I2C parameters to be matched. Both **match.i2c.adapter_id** and **match.i2c.address** should be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.

match.i2c.adapter_id I2C adapter ID to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.

match.i2c.address I2C address to be matched. Used if **match_type** is `V4L2_ASYNC_MATCH_I2C`.

match.custom Driver-specific match criteria. Used if **match_type** is `V4L2_ASYNC_MATCH_CUSTOM`.

match.custom.match Driver-specific match function to be used if `V4L2_ASYNC_MATCH_CUSTOM`.

match.custom.priv Driver-specific private struct with match parameters to be used if `V4L2_ASYNC_MATCH_CUSTOM`.

list used to link struct v4l2_async_subdev objects, waiting to be probed, to a notifier->waiting list

asd_list used to add struct v4l2_async_subdev objects to the master notifier **asd_list**

Description

When this struct is used as a member in a driver specific struct, the driver specific struct shall contain the `struct v4l2_async_subdev` as its first member.

struct **v4l2_async_notifier_operations**
Asynchronous V4L2 notifier operations

Definition

```
struct v4l2_async_notifier_operations {
  int (*bound)(struct v4l2_async_notifier *notifier,struct v4l2_subdev␣
↪*subdev, struct v4l2_async_subdev *asd);
  int (*complete)(struct v4l2_async_notifier *notifier);
  void (*unbind)(struct v4l2_async_notifier *notifier,struct v4l2_subdev␣
↪*subdev, struct v4l2_async_subdev *asd);
};
```

Members

bound a subdevice driver has successfully probed one of the subdevices

complete All subdevices have been probed successfully. The complete callback is only executed for the root notifier.

unbind a subdevice is leaving

struct **v4l2_async_notifier**
v4l2_device notifier data

Definition

```
struct v4l2_async_notifier {
  const struct v4l2_async_notifier_operations *ops;
  struct v4l2_device *v4l2_dev;
  struct v4l2_subdev *sd;
  struct v4l2_async_notifier *parent;
  struct list_head asd_list;
  struct list_head waiting;
  struct list_head done;
  struct list_head list;
};
```

Members

ops notifier operations

v4l2_dev v4l2_device of the root notifier, NULL otherwise

sd sub-device that registered the notifier, NULL otherwise

parent parent notifier

asd_list master list of struct v4l2_async_subdev

waiting list of struct v4l2_async_subdev, waiting for their drivers

done list of struct v4l2_subdev, already probed

list member in a global list of notifiers

void **v4l2_async_notifier_init**(struct v4l2_async_notifier * notifier)
Initialize a notifier.

Parameters

struct v4l2_async_notifier * notifier pointer to `struct v4l2_async_notifier`

Description

This function initializes the notifier **asd_list**. It must be called before the first call to **v4l2_async_notifier_add_subdev**.

int **v4l2_async_notifier_add_subdev**(struct v4l2_async_notifier * notifier,
struct v4l2_async_subdev * asd)
Add an async subdev to the notifier's master asd list.

Parameters

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

struct v4l2_async_subdev * asd pointer to struct v4l2_async_subdev

Description

Call this function before registering a notifier to link the provided asd to the notifiers master **asd_list**.

struct v4l2_async_subdev * **v4l2_async_notifier_add_fwnode_subdev**(struct v4l2_async_notifier * notifier, struct fwnode_handle * fwnode, unsigned int asd_struct_size)

Allocate and add a fwnode async subdev to the notifier's master asd_list.

Parameters

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

struct fwnode_handle * fwnode fwnode handle of the sub-device to be matched

unsigned int asd_struct_size size of the driver's async sub-device struct, including sizeof(struct v4l2_async_subdev). The struct v4l2_async_subdev shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

Description

Allocate a fwnode-matched asd of size asd_struct_size, and add it to the notifiers **asd_list**. The function also gets a reference of the fwnode which is released later at notifier cleanup time.

int **v4l2_async_notifier_add_fwnode_remote_subdev**(struct v4l2_async_notifier * notif, struct fwnode_handle * endpoint, struct v4l2_async_subdev * asd)

Allocate and add a fwnode remote async subdev to the notifier's master asd_list.

Parameters

struct v4l2_async_notifier * notif pointer to struct v4l2_async_notifier

struct fwnode_handle * endpoint local endpoint pointing to the remote sub-device to be matched

struct v4l2_async_subdev * asd Async sub-device struct allocated by the caller. The `struct v4l2_async_subdev` shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

Description

Gets the remote endpoint of a given local endpoint, set it up for fwnode matching and adds the async sub-device to the notifier's **asd_list**. The function also gets a reference of the fwnode which is released later at notifier cleanup time.

This is just like **v4l2_async_notifier_add_fwnode_subdev**, but with the exception that the fwnode refers to a local endpoint, not the remote one, and the function relies on the caller to allocate the async sub-device struct.

struct v4l2_async_subdev * **v4l2_async_notifier_add_i2c_subdev**(struct v4l2_async_notifier * notifier, int adapter_id, unsigned short address, unsigned int asd_struct_size)

Allocate and add an i2c async subdev to the notifier's master asd_list.

Parameters

struct v4l2_async_notifier * notifier pointer to `struct v4l2_async_notifier`

int adapter_id I2C adapter ID to be matched

unsigned short address I2C address of sub-device to be matched

unsigned int asd_struct_size size of the driver's async sub-device struct, including sizeof(struct v4l2_async_subdev). The `struct v4l2_async_subdev` shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

Description

Same as above but for I2C matched sub-devices.

struct v4l2_async_subdev * **v4l2_async_notifier_add_devname_subdev**(struct v4l2_async_notifier * notifier, const char * device_name, unsigned int asd_struct_size)

Allocate and add a device-name async subdev to the notifier's master asd_list.

Parameters

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

const char * device_name device name string to be matched

unsigned int asd_struct_size size of the driver's async sub-device struct, including sizeof(struct v4l2_async_subdev). The struct v4l2_async_subdev shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

Description

Same as above but for device-name matched sub-devices.

int **v4l2_async_notifier_register**(struct v4l2_device * v4l2_dev, struct v4l2_async_notifier * notifier)
registers a subdevice asynchronous notifier

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

int **v4l2_async_subdev_notifier_register**(struct v4l2_subdev * sd, struct v4l2_async_notifier * notifier)
registers a subdevice asynchronous notifier for a sub-device

Parameters

struct v4l2_subdev * sd pointer to struct v4l2_subdev

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

void **v4l2_async_notifier_unregister**(struct v4l2_async_notifier * notifier)
unregisters a subdevice asynchronous notifier

Parameters

struct v4l2_async_notifier * notifier pointer to struct v4l2_async_notifier

void **v4l2_async_notifier_cleanup**(struct v4l2_async_notifier * notifier)
clean up notifier resources

Parameters

struct v4l2_async_notifier * notifier the notifier the resources of which are to be cleaned up

Description

Release memory resources related to a notifier, including the async sub-devices allocated for the purposes of the notifier but not the notifier itself. The user is responsible for calling this function to clean up the notifier after calling **v4l2_async_notifier_add_subdev**, **v4l2_async_notifier_parse_fwnode_endpoints** or **v4l2_fwnode_reference_parse_sensor_c**

There is no harm from calling v4l2_async_notifier_cleanup in other cases as long as its memory has been zeroed after it has been allocated.

int **v4l2_async_register_subdev**(struct v4l2_subdev * sd)
registers a sub-device to the asynchronous subdevice framework

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

int **v4l2_async_register_subdev_sensor_common**(struct v4l2_subdev * sd)
registers a sensor sub-device to the asynchronous sub-device framework and parse set up common sensor related devices

Parameters

struct v4l2_subdev * sd pointer to struct `v4l2_subdev`

Description

This function is just like `v4l2_async_register_subdev()` with the exception that calling it will also parse firmware interfaces for remote references using `v4l2_async_notifier_parse_fwnode_sensor_common()` and registers the async sub-devices. The sub-device is similarly unregistered by calling `v4l2_async_unregister_subdev()`.

While registered, the subdev module is marked as in-use.

An error is returned if the module is no longer loaded on any attempts to register it.

void **v4l2_async_unregister_subdev**(struct v4l2_subdev * sd)
unregisters a sub-device to the asynchronous subdevice framework

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

53.1.23 V4L2 fwnode kAPI

struct **v4l2_fwnode_bus_mipi_csi2**
MIPI CSI-2 bus data structure

Definition

```
struct v4l2_fwnode_bus_mipi_csi2 {
  unsigned int flags;
  unsigned char data_lanes[V4L2_FWNODE_CSI2_MAX_DATA_LANES];
  unsigned char clock_lane;
  unsigned short num_data_lanes;
  bool lane_polarities[1 + V4L2_FWNODE_CSI2_MAX_DATA_LANES];
};
```

Members

flags media bus (V4L2_MBUS_*) flags

data_lanes an array of physical data lane indexes

clock_lane physical lane index of the clock lane

num_data_lanes number of data lanes

lane_polarities polarity of the lanes. The order is the same of the physical lanes.

struct **v4l2_fwnode_bus_parallel**
parallel data bus data structure

Definition

```
struct v4l2_fwnode_bus_parallel {
  unsigned int flags;
  unsigned char bus_width;
  unsigned char data_shift;
};
```

Members

flags media bus (V4L2_MBUS_*) flags

bus_width bus width in bits

data_shift data shift in bits

struct **v4l2_fwnode_bus_mipi_csi1**
CSI-1/CCP2 data bus structure

Definition

```
struct v4l2_fwnode_bus_mipi_csi1 {
  unsigned char clock_inv:1;
  unsigned char strobe:1;
  bool lane_polarity[2];
  unsigned char data_lane;
  unsigned char clock_lane;
};
```

Members

clock_inv polarity of clock/strobe signal false - not inverted, true - inverted

strobe false - data/clock, true - data/strobe

lane_polarity the polarities of the clock (index 0) and data lanes index (1)

data_lane the number of the data lane

clock_lane the number of the clock lane

struct **v4l2_fwnode_endpoint**
the endpoint data structure

Definition

```
struct v4l2_fwnode_endpoint {
  struct fwnode_endpoint base;
  enum v4l2_mbus_type bus_type;
  union {
    struct v4l2_fwnode_bus_parallel parallel;
    struct v4l2_fwnode_bus_mipi_csi1 mipi_csi1;
    struct v4l2_fwnode_bus_mipi_csi2 mipi_csi2;
  } bus;
```

(continues on next page)

(continued from previous page)

```
  u64 *link_frequencies;
  unsigned int nr_of_link_frequencies;
};
```

Members

base fwnode endpoint of the v4l2_fwnode

bus_type bus type

bus union with bus configuration data structure

bus.parallel embedded `struct v4l2_fwnode_bus_parallel`. Used if the bus is parallel.

bus.mipi_csi1 embedded `struct v4l2_fwnode_bus_mipi_csi1`. Used if the bus is MIPI Alliance's Camera Serial Interface version 1 (MIPI CSI1) or Standard Mobile Imaging Architecture's Compact Camera Port 2 (SMIA CCP2).

bus.mipi_csi2 embedded `struct v4l2_fwnode_bus_mipi_csi2`. Used if the bus is MIPI Alliance's Camera Serial Interface version 2 (MIPI CSI2).

link_frequencies array of supported link frequencies

nr_of_link_frequencies number of elements in link_frequenccies array

V4L2_FWNODE_PROPERTY_UNSET()
identify a non initialized property

Parameters

Description

All properties in `struct v4l2_fwnode_device_properties` are initialized to this value.

enum **v4l2_fwnode_orientation**
possible device orientation

Constants

V4L2_FWNODE_ORIENTATION_FRONT device installed on the front side

V4L2_FWNODE_ORIENTATION_BACK device installed on the back side

V4L2_FWNODE_ORIENTATION_EXTERNAL device externally located

struct **v4l2_fwnode_device_properties**
fwnode device properties

Definition

```
struct v4l2_fwnode_device_properties {
  enum v4l2_fwnode_orientation orientation;
  unsigned int rotation;
};
```

Members

orientation device orientation. See `enum v4l2_fwnode_orientation`

rotation device rotation

struct **v4l2_fwnode_link**
a link between two endpoints

Definition

```
struct v4l2_fwnode_link {
  struct fwnode_handle *local_node;
  unsigned int local_port;
  unsigned int local_id;
  struct fwnode_handle *remote_node;
  unsigned int remote_port;
  unsigned int remote_id;
};
```

Members

local_node pointer to device_node of this endpoint

local_port identifier of the port this endpoint belongs to

local_id identifier of the id this endpoint belongs to

remote_node pointer to device_node of the remote endpoint

remote_port identifier of the port the remote endpoint belongs to

remote_id identifier of the id the remote endpoint belongs to

enum **v4l2_connector_type**
connector type

Constants

V4L2_CONN_UNKNOWN unknown connector type, no V4L2 connector configuration

V4L2_CONN_COMPOSITE analog composite connector

V4L2_CONN_SVIDEO analog svideo connector

struct **v4l2_connector_link**
connector link data structure

Definition

```
struct v4l2_connector_link {
  struct list_head head;
  struct v4l2_fwnode_link fwnode_link;
};
```

Members

head structure to be used to add the link to the `struct v4l2_fwnode_connector`

fwnode_link `struct v4l2_fwnode_link` link between the connector and the device the connector belongs to.

struct **v4l2_fwnode_connector_analog**
analog connector data structure

Definition

```
struct v4l2_fwnode_connector_analog {
  v4l2_std_id sdtv_stds;
};
```

Members

sdtv_stds sdtv standards this connector supports, set to V4L2_STD_ALL if no restrictions are specified.

struct **v4l2_fwnode_connector**
the connector data structure

Definition

```
struct v4l2_fwnode_connector {
  const char *name;
  const char *label;
  enum v4l2_connector_type type;
  struct list_head links;
  unsigned int nr_of_links;
  union {
    struct v4l2_fwnode_connector_analog analog;
  } connector;
};
```

Members

name the connector device name

label optional connector label

type connector type

links list of all connector `struct v4l2_connector_link` links

nr_of_links total number of links

connector connector configuration

connector.analog analog connector configuration `struct v4l2_fwnode_connector_analog`

int **v4l2_fwnode_endpoint_parse**(struct fwnode_handle * fwnode, struct v4l2_fwnode_endpoint * vep)
parse all fwnode node properties

Parameters

struct fwnode_handle * fwnode pointer to the endpoint's fwnode handle

struct v4l2_fwnode_endpoint * vep pointer to the V4L2 fwnode data structure

Description

This function parses the V4L2 fwnode endpoint specific parameters from the firmware. The caller is responsible for assigning **vep.bus_type** to a valid media bus type. The caller may also set the default configuration for the endpoint —a configuration that shall be in line with the DT binding documentation. Should a device support multiple bus types, the caller may call this function once the correct type is found —with a default configuration valid for that type.

As a compatibility means guessing the bus type is also supported by setting **vep.bus_type** to V4L2_MBUS_UNKNOWN. The caller may not provide a default configuration in this case as the defaults are specific to a given bus type. This functionality is deprecated and should not be used in new drivers and it is only supported for CSI-2 D-PHY, parallel and Bt.656 buses.

The function does not change the V4L2 fwnode endpoint state if it fails.

NOTE

This function does not parse properties the size of which is variable without a low fixed limit. Please use `v4l2_fwnode_endpoint_alloc_parse()` in new drivers instead.

Return

`0` on success or a negative error code on failure: `-ENOMEM` on memory allocation failure `-EINVAL` on parsing failure `-ENXIO` on mismatching bus types

void **`v4l2_fwnode_endpoint_free`**(struct v4l2_fwnode_endpoint * vep)
free the V4L2 fwnode acquired by `v4l2_fwnode_endpoint_alloc_parse()`

Parameters

`struct v4l2_fwnode_endpoint * vep` the V4L2 fwnode the resources of which are to be released

Description

It is safe to call this function with NULL argument or on a V4L2 fwnode the parsing of which failed.

int **`v4l2_fwnode_endpoint_alloc_parse`**(struct fwnode_handle * fwnode, struct v4l2_fwnode_endpoint * vep)
parse all fwnode node properties

Parameters

`struct fwnode_handle * fwnode` pointer to the endpoint’s fwnode handle

`struct v4l2_fwnode_endpoint * vep` pointer to the V4L2 fwnode data structure

Description

This function parses the V4L2 fwnode endpoint specific parameters from the firmware. The caller is responsible for assigning **vep.bus_type** to a valid media bus type. The caller may also set the default configuration for the endpoint —a configuration that shall be in line with the DT binding documentation. Should a device support multiple bus types, the caller may call this function once the correct type is found —with a default configuration valid for that type.

As a compatibility means guessing the bus type is also supported by setting **vep.bus_type** to V4L2_MBUS_UNKNOWN. The caller may not provide a default configuration in this case as the defaults are specific to a given bus type. This functionality is deprecated and should not be used in new drivers and it is only supported for CSI-2 D-PHY, parallel and Bt.656 buses.

The function does not change the V4L2 fwnode endpoint state if it fails.

`v4l2_fwnode_endpoint_alloc_parse()` has two important differences to `v4l2_fwnode_endpoint_parse()`:

1. It also parses variable size data.
2. The memory it has allocated to store the variable size data must be freed using `v4l2_fwnode_endpoint_free()` when no longer needed.

Return

`0` on success or a negative error code on failure: `-ENOMEM` on memory allocation failure `-EINVAL` on parsing failure `-ENXIO` on mismatching bus types

int **v4l2_fwnode_parse_link**(struct fwnode_handle * fwnode, struct v4l2_fwnode_link * link)
parse a link between two endpoints

Parameters

`struct fwnode_handle * fwnode` pointer to the endpoint's fwnode at the local end of the link

`struct v4l2_fwnode_link * link` pointer to the V4L2 fwnode link data structure

Description

Fill the link structure with the local and remote nodes and port numbers. The local_node and remote_node fields are set to point to the local and remote port's parent nodes respectively (the port parent node being the parent node of the port node if that node isn't a 'ports' node, or the grand-parent node of the port node otherwise).

A reference is taken to both the local and remote nodes, the caller must use `v4l2_fwnode_put_link()` to drop the references when done with the link.

Return

0 on success, or -ENOLINK if the remote endpoint fwnode can't be found.

void **v4l2_fwnode_put_link**(struct v4l2_fwnode_link * link)
drop references to nodes in a link

Parameters

`struct v4l2_fwnode_link * link` pointer to the V4L2 fwnode link data structure

Description

Drop references to the local and remote nodes in the link. This function must be called on every link parsed with `v4l2_fwnode_parse_link()`.

void **v4l2_fwnode_connector_free**(struct v4l2_fwnode_connector * connector)
free the V4L2 connector acquired memory

Parameters

`struct v4l2_fwnode_connector * connector` the V4L2 connector resources of which are to be released

Description

Free all allocated memory and put all links acquired by `v4l2_fwnode_connector_parse()` and `v4l2_fwnode_connector_add_link()`.

It is safe to call this function with NULL argument or on a V4L2 connector the parsing of which failed.

int **v4l2_fwnode_connector_parse**(struct fwnode_handle * fwnode, struct v4l2_fwnode_connector * connector)
initialize the 'struct v4l2_fwnode_connector'

Parameters

struct fwnode_handle * fwnode pointer to the subdev endpoint's fwnode handle where the connector is connected to or to the connector endpoint fwnode handle.

struct v4l2_fwnode_connector * connector pointer to the V4L2 fwnode connector data structure

Description

Fill the `struct v4l2_fwnode_connector` with the connector type, label and all `enum v4l2_connector_type` specific connector data. The label is optional so it is set to `NULL` if no one was found. The function initialize the links to zero. Adding links to the connector is done by calling `v4l2_fwnode_connector_add_link()`.

The memory allocated for the label must be freed when no longer needed. Freeing the memory is done by `v4l2_fwnode_connector_free()`.

Return

- `0` on success or a negative error code on failure:
- `-EINVAL` if **fwnode** is invalid
- `-ENOTCONN` if connector type is unknown or connector device can't be found

int **v4l2_fwnode_connector_add_link**(struct fwnode_handle * fwnode, struct v4l2_fwnode_connector * connector)
add a link between a connector node and a v4l2-subdev node.

Parameters

struct fwnode_handle * fwnode pointer to the subdev endpoint's fwnode handle where the connector is connected to

struct v4l2_fwnode_connector * connector pointer to the V4L2 fwnode connector data structure

Description

Add a new `struct v4l2_connector_link` link to the `struct v4l2_fwnode_connector` connector links list. The link local_node points to the connector node, the remote_node to the host v4l2 (sub)dev.

The taken references to remote_node and local_node must be dropped and the allocated memory must be freed when no longer needed. Both is done by calling `v4l2_fwnode_connector_free()`.

Return

- `0` on success or a negative error code on failure:
- `-EINVAL` if **fwnode** or **connector** is invalid or **connector** type is unknown
- `-ENOMEM` on link memory allocation failure
- `-ENOTCONN` if remote connector device can't be found
- `-ENOLINK` if link parsing between v4l2 (sub)dev and connector fails

int **v4l2_fwnode_device_parse**(struct device * dev, struct v4l2_fwnode_device_properties * props)
parse fwnode device properties

Parameters

struct device * dev pointer to `struct device`

struct v4l2_fwnode_device_properties * props pointer to `struct v4l2_fwnode_device_properties` where to store the parsed properties values

Description

This function parses and validates the V4L2 fwnode device properties from the firmware interface, and fills the **struct** v4l2_fwnode_device_properties provided by the caller.

Return

% 0 on success `-EINVAL` if a parsed property value is not valid

parse_endpoint_func
Typedef: Driver's callback function to be called on each V4L2 fwnode endpoint.

Syntax

```
int parse_endpoint_func (struct device * dev, struct
v4l2_fwnode_endpoint * vep, struct v4l2_async_subdev * asd);
```

Parameters

struct device * dev pointer to `struct device`

struct v4l2_fwnode_endpoint * vep pointer to `struct v4l2_fwnode_endpoint`

struct v4l2_async_subdev * asd pointer to `struct v4l2_async_subdev`

Return

- `0` on success
- `-ENOTCONN` if the endpoint is to be skipped but this should not be considered as an error
- `-EINVAL` if the endpoint configuration is invalid

int **v4l2_async_notifier_parse_fwnode_endpoints**(struct device * dev, struct v4l2_async_notifier * notifier, size_t asd_struct_size, parse_endpoint_func parse_endpoint)

Parse V4L2 fwnode endpoints in a device node

Parameters

struct device * dev the device the endpoints of which are to be parsed

struct v4l2_async_notifier * notifier notifier for **dev**

size_t asd_struct_size size of the driver's async sub-device struct, including sizeof(struct v4l2_async_subdev). The `struct v4l2_async_subdev` shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

parse_endpoint_func parse_endpoint Driver's callback function called on each V4L2 fwnode endpoint. Optional.

Description

Parse the fwnode endpoints of the **dev** device and populate the async sub- devices list in the notifier. The **parse_endpoint** callback function is called for each endpoint with the corresponding async sub-device pointer to let the caller initialize the driver-specific part of the async sub-device structure.

The notifier memory shall be zeroed before this function is called on the notifier.

This function may not be called on a registered notifier and may be called on a notifier only once.

The `struct v4l2_fwnode_endpoint` passed to the callback function **parse_endpoint** is released once the function is finished. If there is a need to retain that configuration, the user needs to allocate memory for it.

Any notifier populated using this function must be released with a call to `v4l2_async_notifier_cleanup()` after it has been unregistered and the async sub-devices are no longer in use, even if the function returned an error.

Return

0 on success, including when no async sub-devices are found `-ENOMEM` if memory allocation failed `-EINVAL` if graph or endpoint parsing failed Other error codes as returned by **parse_endpoint**

int **v4l2_async_notifier_parse_fwnode_endpoints_by_port**(struct device * dev, struct v4l2_async_notifier * notifier, size_t asd_struct_size, unsigned int port, parse_endpoint_func parse_end

Parse V4L2 fwnode endpoints of a port in a device node

Parameters

struct device * dev the device the endpoints of which are to be parsed

struct v4l2_async_notifier * notifier notifier for **dev**

size_t asd_struct_size size of the driver's async sub-device struct, including sizeof(struct v4l2_async_subdev). The `struct v4l2_async_subdev` shall be the first member of the driver's async sub-device struct, i.e. both begin at the same memory address.

unsigned int port port number where endpoints are to be parsed

parse_endpoint_func parse_endpoint Driver's callback function called on each V4L2 fwnode endpoint. Optional.

Description

This function is just like `v4l2_async_notifier_parse_fwnode_endpoints()` with the exception that it only parses endpoints in a given port. This is useful on devices that have both sinks and sources: the async sub-devices connected to sources have already been configured by another driver (on capture devices). In this case the driver must know which ports to parse.

Parse the fwnode endpoints of the **dev** device on a given **port** and populate the async sub-devices list of the notifier. The **parse_endpoint** callback function is called for each endpoint with the corresponding async sub-device pointer to let the caller initialize the driver-specific part of the async sub-device structure.

The notifier memory shall be zeroed before this function is called on the notifier the first time.

This function may not be called on a registered notifier and may be called on a notifier only once per port.

The `struct v4l2_fwnode_endpoint` passed to the callback function **parse_endpoint** is released once the function is finished. If there is a need to retain that configuration, the user needs to allocate memory for it.

Any notifier populated using this function must be released with a call to `v4l2_async_notifier_cleanup()` after it has been unregistered and the async sub-devices are no longer in use, even if the function returned an error.

Return

0 on success, including when no async sub-devices are found `-ENOMEM` if memory allocation failed `-EINVAL` if graph or endpoint parsing failed Other error codes as returned by **parse_endpoint**

int **v4l2_async_notifier_parse_fwnode_sensor_common**(struct device * dev, struct v4l2_async_notifier * notifier)

parse common references on sensors for async sub-devices

Parameters

struct device * dev the device node the properties of which are parsed for references

struct v4l2_async_notifier * notifier the async notifier where the async subdevs will be added

Description

Parse common sensor properties for remote devices related to the sensor and set up async sub-devices for them.

Any notifier populated using this function must be released with a call to v4l2_async_notifier_release() after it has been unregistered and the async sub-devices are no longer in use, even in the case the function returned an error.

Return

0 on success -ENOMEM if memory allocation failed -EINVAL if property parsing failed

53.1.24 V4L2 rect helper functions

void **v4l2_rect_set_size_to**(struct v4l2_rect * r, const struct v4l2_rect * size)
copy the width/height values.

Parameters

struct v4l2_rect * r rect whose width and height fields will be set

const struct v4l2_rect * size rect containing the width and height fields you need.

void **v4l2_rect_set_min_size**(struct v4l2_rect * r, const struct v4l2_rect * min_size)
width and height of r should be >= min_size.

Parameters

struct v4l2_rect * r rect whose width and height will be modified

const struct v4l2_rect * min_size rect containing the minimal width and height

void **v4l2_rect_set_max_size**(struct v4l2_rect * r, const struct v4l2_rect * max_size)
width and height of r should be <= max_size

Parameters

struct v4l2_rect * r rect whose width and height will be modified

const struct v4l2_rect * max_size rect containing the maximum width and height

void **v4l2_rect_map_inside**(struct v4l2_rect * r, const struct v4l2_rect * boundary)
r should be inside boundary.

Parameters

struct v4l2_rect * r rect that will be modified

const struct v4l2_rect * boundary rect containing the boundary for **r**

bool **v4l2_rect_same_size**(const struct v4l2_rect * r1, const struct v4l2_rect * r2)
return true if r1 has the same size as r2

Parameters

const struct v4l2_rect * r1 rectangle.

const struct v4l2_rect * r2 rectangle.

Description

Return true if both rectangles have the same size.

bool **v4l2_rect_same_position**(const struct v4l2_rect * r1, const struct v4l2_rect * r2)
return true if r1 has the same position as r2

Parameters

const struct v4l2_rect * r1 rectangle.

const struct v4l2_rect * r2 rectangle.

Description

Return true if both rectangles have the same position

bool **v4l2_rect_equal**(const struct v4l2_rect * r1, const struct v4l2_rect * r2)
return true if r1 equals r2

Parameters

const struct v4l2_rect * r1 rectangle.

const struct v4l2_rect * r2 rectangle.

Description

Return true if both rectangles have the same size and position.

void **v4l2_rect_intersect**(struct v4l2_rect * r, const struct v4l2_rect * r1, const struct v4l2_rect * r2)
calculate the intersection of two rects.

Parameters

struct v4l2_rect * r intersection of **r1** and **r2**.

const struct v4l2_rect * r1 rectangle.

const struct v4l2_rect * r2 rectangle.

void **v4l2_rect_scale**(struct v4l2_rect * r, const struct v4l2_rect * from, const struct v4l2_rect * to)
scale rect r by to/from

Parameters

struct v4l2_rect * r rect to be scaled.

const struct v4l2_rect * from from rectangle.

const struct v4l2_rect * to to rectangle.

Description

This scales rectangle **r** horizontally by **to->width** / **from->width** and vertically by **to->height** / **from->height**.

Typically **r** is a rectangle inside **from** and you want the rectangle as it would appear after scaling **from** to **to**. So the resulting **r** will be the scaled rectangle inside **to**.

bool **v4l2_rect_overlap**(const struct v4l2_rect * r1, const struct v4l2_rect * r2)
do r1 and r2 overlap?

Parameters

const struct v4l2_rect * r1 rectangle.

const struct v4l2_rect * r2 rectangle.

Description

Returns true if **r1** and **r2** overlap.

53.1.25 Tuner functions and data structures

enum **tuner_mode**
Mode of the tuner

Constants

T_RADIO Tuner core will work in radio mode

T_ANALOG_TV Tuner core will work in analog TV mode

Description

Older boards only had a single tuner device, but some devices have a separate tuner for radio. In any case, the tuner-core needs to know if the tuner chip(s) will be used in radio mode or analog TV mode, as, on radio mode, frequencies are specified on a different range than on TV mode. This enum is used by the tuner core in order to work with the proper tuner range and eventually use a different tuner chip while in radio mode.

struct **tuner_setup**
setup the tuner chipsets

Definition

```
struct tuner_setup {
  unsigned short  addr;
  unsigned int    type;
  unsigned int    mode_mask;
  void *config;
  int (*tuner_callback)(void *dev, int component, int cmd, int arg);
};
```

Members

addr I2C address used to control the tuner device/chipset

type Type of the tuner, as defined at the TUNER_* macros. Each different tuner model should have an unique identifier.

mode_mask Mask with the allowed tuner modes: V4L2_TUNER_RADIO, V4L2_TUNER_ANALOG_TV and/or V4L2_TUNER_DIGITAL_TV, describing if the tuner should be used to support Radio, analog TV and/or digital TV.

config Used to send tuner-specific configuration for complex tuners that require extra parameters to be set. Only a very few tuners require it and its usage on newer tuners should be avoided.

tuner_callback Some tuners require to call back the bridge driver, in order to do some tasks like rising a GPIO at the bridge chipset, in order to do things like resetting the device.

Description

Older boards only had a single tuner device. Nowadays multiple tuner devices may be present on a single board. Using TUNER_SET_TYPE_ADDR to pass the tuner_setup structure it is possible to setup each tuner device in turn.

Since multiple devices may be present it is no longer sufficient to send a command to a single i2c device. Instead you should broadcast the command to all i2c devices.

By setting the mode_mask correctly you can select which commands are accepted by a specific tuner device. For example, set mode_mask to T_RADIO if the device is a radio-only tuner. That specific tuner will only accept commands when the tuner is in radio mode and ignore them when the tuner is set to TV mode.

enum **param_type**
type of the tuner pameters

Constants

TUNER_PARAM_TYPE_RADIO Tuner params are for FM and/or AM radio

TUNER_PARAM_TYPE_PAL Tuner params are for PAL color TV standard

TUNER_PARAM_TYPE_SECAM Tuner params are for SECAM color TV standard

TUNER_PARAM_TYPE_NTSC Tuner params are for NTSC color TV standard

TUNER_PARAM_TYPE_DIGITAL Tuner params are for digital TV

struct **tuner_range**
define the frequencies supported by the tuner

Definition

```
struct tuner_range {
  unsigned short limit;
  unsigned char config;
  unsigned char cb;
};
```

Members

limit Max frequency supported by that range, in 62.5 kHz (TV) or 62.5 Hz (Radio), as defined by V4L2_TUNER_CAP_LOW.

config Value of the band switch byte (BB) to setup this mode.

cb Value of the CB byte to setup this mode.

Description

Please notice that digital tuners like xc3028/xc4000/xc5000 don't use those ranges, as they're defined inside the driver. This is used by analog tuners that are compatible with the "Philips way" to setup the tuners. On those devices, the tuner set is done via 4 bytes:

1) divider byte1 (DB1)
2) divider byte 2 (DB2)
3) Control byte (CB)
4) band switch byte (BB)

Some tuners also have an additional optional Auxiliary byte (AB).

struct **tuner_params**

Parameters to be used to setup the tuner. Those are used by drivers/media/tuners/tuner-types.c in order to specify the tuner properties. Most of the parameters are for tuners based on tda9887 IF-PLL multi-standard analog TV/Radio demodulator, with is very common on legacy analog tuners.

Definition

```
struct tuner_params {
  enum param_type type;
  unsigned int cb_first_if_lower_freq:1;
  unsigned int has_tda9887:1;
  unsigned int port1_fm_high_sensitivity:1;
  unsigned int port2_fm_high_sensitivity:1;
  unsigned int fm_gain_normal:1;
  unsigned int intercarrier_mode:1;
  unsigned int port1_active:1;
  unsigned int port2_active:1;
  unsigned int port1_invert_for_secam_lc:1;
  unsigned int port2_invert_for_secam_lc:1;
  unsigned int port1_set_for_fm_mono:1;
  unsigned int default_pll_gating_18:1;
  unsigned int radio_if:2;
  signed int default_top_low:5;
  signed int default_top_mid:5;
  signed int default_top_high:5;
  signed int default_top_secam_low:5;
  signed int default_top_secam_mid:5;
  signed int default_top_secam_high:5;
  u16 iffreq;
  unsigned int count;
  struct tuner_range *ranges;
};
```

Members

type Type of the tuner parameters, as defined at enum param_type. If the tuner

supports multiple standards, an array should be used, with one row per different standard.

cb_first_if_lower_freq Many Philips-based tuners have a comment in their datasheet like "For channel selection involving band switching, and to ensure smooth tuning to the desired channel without causing unnecessary charge pump action, it is recommended to consider the difference between wanted channel frequency and the current channel frequency. Unnecessary charge pump action will result in very low tuning voltage which may drive the oscillator to extreme conditions". Set cb_first_if_lower_freq to 1, if this check is required for this tuner. I tested this for PAL by first setting the TV frequency to 203 MHz and then switching to 96.6 MHz FM radio. The result was static unless the control byte was sent first.

has_tda9887 Set to 1 if this tuner uses a tda9887

port1_fm_high_sensitivity Many Philips tuners use tda9887 PORT1 to select the FM radio sensitivity. If this setting is 1, then set PORT1 to 1 to get proper FM reception.

port2_fm_high_sensitivity Some Philips tuners use tda9887 PORT2 to select the FM radio sensitivity. If this setting is 1, then set PORT2 to 1 to get proper FM reception.

fm_gain_normal Some Philips tuners use tda9887 cGainNormal to select the FM radio sensitivity. If this setting is 1, e register will use cGainNormal instead of cGainLow.

intercarrier_mode Most tuners with a tda9887 use QSS mode. Some (cheaper) tuners use Intercarrier mode. If this setting is 1, then the tuner needs to be set to intercarrier mode.

port1_active This setting sets the default value for PORT1. 0 means inactive, 1 means active. Note: the actual bit value written to the tda9887 is inverted. So a 0 here means a 1 in the B6 bit.

port2_active This setting sets the default value for PORT2. 0 means inactive, 1 means active. Note: the actual bit value written to the tda9887 is inverted. So a 0 here means a 1 in the B7 bit.

port1_invert_for_secam_lc Sometimes PORT1 is inverted when the SECAM-L' standard is selected. Set this bit to 1 if this is needed.

port2_invert_for_secam_lc Sometimes PORT2 is inverted when the SECAM-L' standard is selected. Set this bit to 1 if this is needed.

port1_set_for_fm_mono Some cards require PORT1 to be 1 for mono Radio FM and 0 for stereo.

default_pll_gating_18 Select 18% (or according to datasheet 0%) L standard PLL gating, vs the driver default of 36%.

radio_if IF to use in radio mode. Tuners with a separate radio IF filter seem to use 10.7, while those without use 33.3 for PAL/SECAM tuners and 41.3 for NTSC tuners. 0 = 10.7, 1 = 33.3, 2 = 41.3

default_top_low Default tda9887 TOP value in dB for the low band. Default is 0. Range: -16:+15

default_top_mid Default tda9887 TOP value in dB for the mid band. Default is 0. Range: -16:+15

default_top_high Default tda9887 TOP value in dB for the high band. Default is 0. Range: -16:+15

default_top_secam_low Default tda9887 TOP value in dB for SECAM-L/L' for the low band. Default is 0. Several tuners require a different TOP value for the SECAM-L/L' standards. Range: -16:+15

default_top_secam_mid Default tda9887 TOP value in dB for SECAM-L/L' for the mid band. Default is 0. Several tuners require a different TOP value for the SECAM-L/L' standards. Range: -16:+15

default_top_secam_high Default tda9887 TOP value in dB for SECAM-L/L' for the high band. Default is 0. Several tuners require a different TOP value for the SECAM-L/L' standards. Range: -16:+15

iffreq Intermediate frequency (IF) used by the tuner on digital mode.

count Size of the ranges array.

ranges Array with the frequency ranges supported by the tuner.

struct **tunertype**
describes the known tuners.

Definition

```
struct tunertype {
  char *name;
  unsigned int count;
  struct tuner_params *params;
  u16 min;
  u16 max;
  u32 stepsize;
  u8 *initdata;
  u8 *sleepdata;
};
```

Members

name string with the tuner's name.

count size of `struct tuner_params` array.

params pointer to `struct tuner_params` array.

min minimal tuner frequency, in 62.5 kHz step. should be multiplied to 16 to convert to MHz.

max minimal tuner frequency, in 62.5 kHz step. Should be multiplied to 16 to convert to MHz.

stepsize frequency step, in Hz.

initdata optional byte sequence to initialize the tuner.

sleepdata optional byte sequence to power down the tuner.

53.1.26 V4L2 common functions and data structures

int **v4l2_ctrl_query_fill**(struct v4l2_queryctrl * qctrl, s32 min, s32 max, s32 step, s32 def)
Fill in a struct v4l2_queryctrl

Parameters

struct v4l2_queryctrl * qctrl pointer to the `struct v4l2_queryctrl` to be filled

s32 min minimum value for the control

s32 max maximum value for the control

s32 step control step

s32 def default value for the control

Description

Fills the `struct v4l2_queryctrl` fields for the query control.

Note: This function assumes that the **qctrl->id** field is filled.

Returns -EINVAL if the control is not known by the V4L2 core, 0 on success.

enum **v4l2_i2c_tuner_type**
specifies the range of tuner address that should be used when seeking for I2C devices.

Constants

ADDRS_RADIO Radio tuner addresses. Represent the following I2C addresses: 0x10 (if compiled with tea5761 support) and 0x60.

ADDRS_DEMOD Demod tuner addresses. Represent the following I2C addresses: 0x42, 0x43, 0x4a and 0x4b.

ADDRS_TV TV tuner addresses. Represent the following I2C addresses: 0x42, 0x43, 0x4a, 0x4b, 0x60, 0x61, 0x62, 0x63 and 0x64.

ADDRS_TV_WITH_DEMOD TV tuner addresses if demod is present, this excludes addresses used by the demodulator from the list of candidates. Represent the following I2C addresses: 0x60, 0x61, 0x62, 0x63 and 0x64.

NOTE

All I2C addresses above use the 7-bit notation.

struct v4l2_subdev * **v4l2_i2c_new_subdev**(struct v4l2_device * v4l2_dev, struct i2c_adapter * adapter, const char * client_type, u8 addr, const unsigned short * probe_addrs)
Load an i2c module and return an initialized `struct v4l2_subdev`.

Parameters

struct v4l2_device * v4l2_dev pointer to `struct v4l2_device`

struct i2c_adapter * adapter pointer to struct i2c_adapter

const char * client_type name of the chip that's on the adapter.

u8 addr I2C address. If zero, it will use **probe_addrs**

const unsigned short * probe_addrs array with a list of address. The last entry at such array should be `I2C_CLIENT_END`.

Description

returns a `struct v4l2_subdev` pointer.

struct v4l2_subdev * **v4l2_i2c_new_subdev_board**(struct v4l2_device * v4l2_dev, struct i2c_adapter * adapter, struct i2c_board_info * info, const unsigned short * probe_addrs)

Load an i2c module and return an initialized `struct v4l2_subdev`.

Parameters

struct v4l2_device * v4l2_dev pointer to `struct v4l2_device`

struct i2c_adapter * adapter pointer to struct i2c_adapter

struct i2c_board_info * info pointer to struct i2c_board_info used to replace the irq, platform_data and addr arguments.

const unsigned short * probe_addrs array with a list of address. The last entry at such array should be `I2C_CLIENT_END`.

Description

returns a `struct v4l2_subdev` pointer.

void **v4l2_i2c_subdev_set_name**(struct v4l2_subdev * sd, struct i2c_client * client, const char * devname, const char * postfix)

Set name for an I^{2}C sub-device

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

struct i2c_client * client pointer to struct i2c_client

const char * devname the name of the device; if NULL, the I^{2}C device's name will be used

const char * postfix sub-device specific string to put right after the I^{2}C device name; may be NULL

void **v4l2_i2c_subdev_init**(struct v4l2_subdev * sd, struct i2c_client * client, const struct v4l2_subdev_ops * ops)

Initializes a `struct v4l2_subdev` with data from an i2c_client struct.

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

struct i2c_client * client pointer to struct i2c_client

const struct v4l2_subdev_ops * ops pointer to struct v4l2_subdev_ops

unsigned short **v4l2_i2c_subdev_addr**(struct v4l2_subdev * sd)
returns i2c client address of struct v4l2_subdev.

Parameters

struct v4l2_subdev * sd pointer to struct v4l2_subdev

Description

Returns the address of an I2C sub-device

const unsigned short * **v4l2_i2c_tuner_addrs**(enum v4l2_i2c_tuner_type type)
Return a list of I2C tuner addresses to probe.

Parameters

enum v4l2_i2c_tuner_type type type of the tuner to seek, as defined by enum v4l2_i2c_tuner_type.

NOTE

Use only if the tuner addresses are unknown.

void **v4l2_i2c_subdev_unregister**(struct v4l2_subdev * sd)
Unregister a v4l2_subdev

Parameters

struct v4l2_subdev * sd pointer to struct v4l2_subdev

struct v4l2_subdev * **v4l2_spi_new_subdev**(struct v4l2_device * v4l2_dev, struct spi_master * master, struct spi_board_info * info)
Load an spi module and return an initialized struct v4l2_subdev.

Parameters

struct v4l2_device * v4l2_dev pointer to struct v4l2_device.

struct spi_master * master pointer to struct spi_master.

struct spi_board_info * info pointer to struct spi_board_info.

Description

returns a struct v4l2_subdev pointer.

void **v4l2_spi_subdev_init**(struct v4l2_subdev * sd, struct spi_device * spi, const struct v4l2_subdev_ops * ops)
Initialize a v4l2_subdev with data from an spi_device struct.

Parameters

struct v4l2_subdev * sd pointer to struct v4l2_subdev

struct spi_device * spi pointer to struct spi_device.

const struct v4l2_subdev_ops * ops pointer to struct v4l2_subdev_ops

void **v4l2_spi_subdev_unregister**(struct v4l2_subdev * sd)
Unregister a v4l2_subdev

Parameters

struct v4l2_subdev * sd pointer to `struct v4l2_subdev`

void **v4l_bound_align_image**(unsigned int * width, unsigned int wmin, unsigned int wmax, unsigned int walign, unsigned int * height, unsigned int hmin, unsigned int hmax, unsigned int halign, unsigned int salign)

adjust video dimensions according to a given constraints.

Parameters

unsigned int * width pointer to width that will be adjusted if needed.

unsigned int wmin minimum width.

unsigned int wmax maximum width.

unsigned int walign least significant bit on width.

unsigned int * height pointer to height that will be adjusted if needed.

unsigned int hmin minimum height.

unsigned int hmax maximum height.

unsigned int halign least significant bit on height.

unsigned int salign least significant bit for the image size (e. g. $width * height$).

Description

Clip an image to have **width** between **wmin** and **wmax**, and **height** between **hmin** and **hmax**, inclusive.

Additionally, the **width** will be a multiple of 2^{walign}, the **height** will be a multiple of 2^{halign}, and the overall size $width * height$ will be a multiple of 2^{salign}.

Note:

1. The clipping rectangle may be shrunk or enlarged to fit the alignment constraints.
2. **wmax** must not be smaller than **wmin**.
3. **hmax** must not be smaller than **hmin**.
4. The alignments must not be so high there are no possible image sizes within the allowed bounds.
5. **wmin** and **hmin** must be at least 1 (don't use 0).
6. For **walign**, **halign** and **salign**, if you don't care about a certain alignment, specify `0`, as $2^0 = 1$ and one byte alignment is equivalent to no alignment.
7. If you only want to adjust downward, specify a maximum that's the same as the initial value.

---

v4l2_find_nearest_size(array, array_size, width_field, height_field, width, height)

Find the nearest size among a discrete set of resolutions contained in an array of a driver specific struct.

Parameters

array a driver specific array of image sizes

array_size the length of the driver specific array of image sizes

width_field the name of the width field in the driver specific struct

height_field the name of the height field in the driver specific struct

width desired width.

height desired height.

Description

Finds the closest resolution to minimize the width and height differences between what requested and the supported resolutions. The size of the width and height fields in the driver specific must equal to that of u32, i.e. four bytes.

Returns the best match or NULL if the length of the array is zero.

int **v4l2_g_parm_cap**(struct video_device * vdev, struct v4l2_subdev * sd, struct v4l2_streamparm * a)

helper routine for vidioc_g_parm to fill this in by calling the g_frame_interval op of the given subdev. It only works for V4L2_BUF_TYPE_VIDEO_CAPTURE(_MPLANE), hence the _cap in the function name.

Parameters

struct video_device * vdev the struct video_device pointer. Used to determine the device caps.

struct v4l2_subdev * sd the sub-device pointer.

struct v4l2_streamparm * a the VIDIOC_G_PARM argument.

int **v4l2_s_parm_cap**(struct video_device * vdev, struct v4l2_subdev * sd, struct v4l2_streamparm * a)

helper routine for vidioc_s_parm to fill this in by calling the s_frame_interval op of the given subdev. It only works for V4L2_BUF_TYPE_VIDEO_CAPTURE(_MPLANE), hence the _cap in the function name.

Parameters

struct video_device * vdev the struct video_device pointer. Used to determine the device caps.

struct v4l2_subdev * sd the sub-device pointer.

struct v4l2_streamparm * a the VIDIOC_S_PARM argument.

enum **v4l2_pixel_encoding**

specifies the pixel encoding value

Constants

V4L2_PIXEL_ENC_UNKNOWN Pixel encoding is unknown/un-initialized

V4L2_PIXEL_ENC_YUV Pixel encoding is YUV

V4L2_PIXEL_ENC_RGB Pixel encoding is RGB

V4L2_PIXEL_ENC_BAYER Pixel encoding is Bayer

struct **v4l2_format_info**
information about a V4L2 format

Definition

```
struct v4l2_format_info {
  u32 format;
  u8 pixel_enc;
  u8 mem_planes;
  u8 comp_planes;
  u8 bpp[4];
  u8 hdiv;
  u8 vdiv;
  u8 block_w[4];
  u8 block_h[4];
};
```

Members

format 4CC format identifier (V4L2_PIX_FMT_*)

pixel_enc Pixel encoding (see enum v4l2_pixel_encoding above)

mem_planes Number of memory planes, which includes the alpha plane (1 to 4).

comp_planes Number of component planes, which includes the alpha plane (1 to 4).

bpp Array of per-plane bytes per pixel

hdiv Horizontal chroma subsampling factor

vdiv Vertical chroma subsampling factor

block_w Per-plane macroblock pixel width (optional)

block_h Per-plane macroblock pixel height (optional)

struct **v4l2_ioctl_ops**
describe operations for each V4L2 ioctl

Definition

```
struct v4l2_ioctl_ops {
  int (*vidioc_querycap)(struct file *file, void *fh, struct v4l2_
↪capability *cap);
  int (*vidioc_enum_fmt_vid_cap)(struct file *file, void *fh, struct v4l2_
↪fmtdesc *f);
  int (*vidioc_enum_fmt_vid_overlay)(struct file *file, void *fh, struct␣
↪v4l2_fmtdesc *f);
  int (*vidioc_enum_fmt_vid_out)(struct file *file, void *fh, struct v4l2_
↪fmtdesc *f);
  int (*vidioc_enum_fmt_sdr_cap)(struct file *file, void *fh, struct v4l2_
↪fmtdesc *f);
```

(continues on next page)

(continued from previous page)

```
  int (*vidioc_enum_fmt_sdr_out)(struct file *file, void *fh, struct v4l2_
↪fmtdesc *f);
  int (*vidioc_enum_fmt_meta_cap)(struct file *file, void *fh, struct v4l2_
↪fmtdesc *f);
  int (*vidioc_enum_fmt_meta_out)(struct file *file, void *fh, struct v4l2_
↪fmtdesc *f);
  int (*vidioc_g_fmt_vid_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_g_fmt_vid_overlay)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_g_fmt_vid_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_g_fmt_vid_out_overlay)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_g_fmt_vbi_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_g_fmt_vbi_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_g_fmt_sliced_vbi_cap)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_g_fmt_sliced_vbi_out)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_g_fmt_vid_cap_mplane)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_g_fmt_vid_out_mplane)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_g_fmt_sdr_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_g_fmt_sdr_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_g_fmt_meta_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_g_fmt_meta_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_s_fmt_vid_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_s_fmt_vid_overlay)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_s_fmt_vid_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_s_fmt_vid_out_overlay)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_s_fmt_vbi_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_s_fmt_vbi_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_s_fmt_sliced_vbi_cap)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_s_fmt_sliced_vbi_out)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_s_fmt_vid_cap_mplane)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_s_fmt_vid_out_mplane)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_s_fmt_sdr_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
```

(continues on next page)

(continued from previous page)

```
  int (*vidioc_s_fmt_sdr_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_s_fmt_meta_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_s_fmt_meta_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_try_fmt_vid_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_try_fmt_vid_overlay)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_try_fmt_vid_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_try_fmt_vid_out_overlay)(struct file *file, void *fh,␣
↪struct v4l2_format *f);
  int (*vidioc_try_fmt_vbi_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_try_fmt_vbi_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_try_fmt_sliced_vbi_cap)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_try_fmt_sliced_vbi_out)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_try_fmt_vid_cap_mplane)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_try_fmt_vid_out_mplane)(struct file *file, void *fh, struct␣
↪v4l2_format *f);
  int (*vidioc_try_fmt_sdr_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_try_fmt_sdr_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_try_fmt_meta_cap)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_try_fmt_meta_out)(struct file *file, void *fh, struct v4l2_
↪format *f);
  int (*vidioc_reqbufs)(struct file *file, void *fh, struct v4l2_
↪requestbuffers *b);
  int (*vidioc_querybuf)(struct file *file, void *fh, struct v4l2_buffer␣
↪*b);
  int (*vidioc_qbuf)(struct file *file, void *fh, struct v4l2_buffer *b);
  int (*vidioc_expbuf)(struct file *file, void *fh, struct v4l2_
↪exportbuffer *e);
  int (*vidioc_dqbuf)(struct file *file, void *fh, struct v4l2_buffer *b);
  int (*vidioc_create_bufs)(struct file *file, void *fh, struct v4l2_
↪create_buffers *b);
  int (*vidioc_prepare_buf)(struct file *file, void *fh, struct v4l2_
↪buffer *b);
  int (*vidioc_overlay)(struct file *file, void *fh, unsigned int i);
  int (*vidioc_g_fbuf)(struct file *file, void *fh, struct v4l2_
↪framebuffer *a);
  int (*vidioc_s_fbuf)(struct file *file, void *fh, const struct v4l2_
↪framebuffer *a);
  int (*vidioc_streamon)(struct file *file, void *fh, enum v4l2_buf_type␣
↪i);
  int (*vidioc_streamoff)(struct file *file, void *fh, enum v4l2_buf_type␣
↪i);
  int (*vidioc_g_std)(struct file *file, void *fh, v4l2_std_id *norm);
```

(continues on next page)

(continued from previous page)

```
  int (*vidioc_s_std)(struct file *file, void *fh, v4l2_std_id norm);
  int (*vidioc_querystd)(struct file *file, void *fh, v4l2_std_id *a);
  int (*vidioc_enum_input)(struct file *file, void *fh, struct v4l2_input␣
↪*inp);
  int (*vidioc_g_input)(struct file *file, void *fh, unsigned int *i);
  int (*vidioc_s_input)(struct file *file, void *fh, unsigned int i);
  int (*vidioc_enum_output)(struct file *file, void *fh, struct v4l2_
↪output *a);
  int (*vidioc_g_output)(struct file *file, void *fh, unsigned int *i);
  int (*vidioc_s_output)(struct file *file, void *fh, unsigned int i);
  int (*vidioc_queryctrl)(struct file *file, void *fh, struct v4l2_
↪queryctrl *a);
  int (*vidioc_query_ext_ctrl)(struct file *file, void *fh, struct v4l2_
↪query_ext_ctrl *a);
  int (*vidioc_g_ctrl)(struct file *file, void *fh, struct v4l2_control␣
↪*a);
  int (*vidioc_s_ctrl)(struct file *file, void *fh, struct v4l2_control␣
↪*a);
  int (*vidioc_g_ext_ctrls)(struct file *file, void *fh, struct v4l2_ext_
↪controls *a);
  int (*vidioc_s_ext_ctrls)(struct file *file, void *fh, struct v4l2_ext_
↪controls *a);
  int (*vidioc_try_ext_ctrls)(struct file *file, void *fh, struct v4l2_ext_
↪controls *a);
  int (*vidioc_querymenu)(struct file *file, void *fh, struct v4l2_
↪querymenu *a);
  int (*vidioc_enumaudio)(struct file *file, void *fh, struct v4l2_audio␣
↪*a);
  int (*vidioc_g_audio)(struct file *file, void *fh, struct v4l2_audio *a);
  int (*vidioc_s_audio)(struct file *file, void *fh, const struct v4l2_
↪audio *a);
  int (*vidioc_enumaudout)(struct file *file, void *fh, struct v4l2_
↪audioout *a);
  int (*vidioc_g_audout)(struct file *file, void *fh, struct v4l2_audioout␣
↪*a);
  int (*vidioc_s_audout)(struct file *file, void *fh, const struct v4l2_
↪audioout *a);
  int (*vidioc_g_modulator)(struct file *file, void *fh, struct v4l2_
↪modulator *a);
  int (*vidioc_s_modulator)(struct file *file, void *fh, const struct v4l2_
↪modulator *a);
  int (*vidioc_g_pixelaspect)(struct file *file, void *fh, int buf_type,␣
↪struct v4l2_fract *aspect);
  int (*vidioc_g_selection)(struct file *file, void *fh, struct v4l2_
↪selection *s);
  int (*vidioc_s_selection)(struct file *file, void *fh, struct v4l2_
↪selection *s);
  int (*vidioc_g_jpegcomp)(struct file *file, void *fh, struct v4l2_
↪jpegcompression *a);
  int (*vidioc_s_jpegcomp)(struct file *file, void *fh, const struct v4l2_
↪jpegcompression *a);
  int (*vidioc_g_enc_index)(struct file *file, void *fh, struct v4l2_enc_
↪idx *a);
  int (*vidioc_encoder_cmd)(struct file *file, void *fh, struct v4l2_
↪encoder_cmd *a);
  int (*vidioc_try_encoder_cmd)(struct file *file, void *fh, struct v4l2_
↪encoder_cmd *a);
```

(continues on next page)

(continued from previous page)

```
  int (*vidioc_decoder_cmd)(struct file *file, void *fh, struct v4l2_
↪decoder_cmd *a);
  int (*vidioc_try_decoder_cmd)(struct file *file, void *fh, struct v4l2_
↪decoder_cmd *a);
  int (*vidioc_g_parm)(struct file *file, void *fh, struct v4l2_streamparm␣
↪*a);
  int (*vidioc_s_parm)(struct file *file, void *fh, struct v4l2_streamparm␣
↪*a);
  int (*vidioc_g_tuner)(struct file *file, void *fh, struct v4l2_tuner *a);
  int (*vidioc_s_tuner)(struct file *file, void *fh, const struct v4l2_
↪tuner *a);
  int (*vidioc_g_frequency)(struct file *file, void *fh, struct v4l2_
↪frequency *a);
  int (*vidioc_s_frequency)(struct file *file, void *fh, const struct v4l2_
↪frequency *a);
  int (*vidioc_enum_freq_bands)(struct file *file, void *fh, struct v4l2_
↪frequency_band *band);
  int (*vidioc_g_sliced_vbi_cap)(struct file *file, void *fh, struct v4l2_
↪sliced_vbi_cap *a);
  int (*vidioc_log_status)(struct file *file, void *fh);
  int (*vidioc_s_hw_freq_seek)(struct file *file, void *fh, const struct␣
↪v4l2_hw_freq_seek *a);
#ifdef CONFIG_VIDEO_ADV_DEBUG;
  int (*vidioc_g_register)(struct file *file, void *fh, struct v4l2_dbg_
↪register *reg);
  int (*vidioc_s_register)(struct file *file, void *fh, const struct v4l2_
↪dbg_register *reg);
  int (*vidioc_g_chip_info)(struct file *file, void *fh, struct v4l2_dbg_
↪chip_info *chip);
#endif;
  int (*vidioc_enum_framesizes)(struct file *file, void *fh, struct v4l2_
↪frmsizeenum *fsize);
  int (*vidioc_enum_frameintervals)(struct file *file, void *fh, struct␣
↪v4l2_frmivalenum *fival);
  int (*vidioc_s_dv_timings)(struct file *file, void *fh, struct v4l2_dv_
↪timings *timings);
  int (*vidioc_g_dv_timings)(struct file *file, void *fh, struct v4l2_dv_
↪timings *timings);
  int (*vidioc_query_dv_timings)(struct file *file, void *fh, struct v4l2_
↪dv_timings *timings);
  int (*vidioc_enum_dv_timings)(struct file *file, void *fh, struct v4l2_
↪enum_dv_timings *timings);
  int (*vidioc_dv_timings_cap)(struct file *file, void *fh, struct v4l2_dv_
↪timings_cap *cap);
  int (*vidioc_g_edid)(struct file *file, void *fh, struct v4l2_edid␣
↪*edid);
  int (*vidioc_s_edid)(struct file *file, void *fh, struct v4l2_edid␣
↪*edid);
  int (*vidioc_subscribe_event)(struct v4l2_fh *fh, const struct v4l2_
↪event_subscription *sub);
  int (*vidioc_unsubscribe_event)(struct v4l2_fh *fh, const struct v4l2_
↪event_subscription *sub);
  long (*vidioc_default)(struct file *file, void *fh, bool valid_prio,␣
↪unsigned int cmd, void *arg);
};
```

Members

vidioc_querycap pointer to the function that implements VIDIOC_QUERYCAP ioctl

vidioc_enum_fmt_vid_cap pointer to the function that implements VIDIOC_ENUM_FMT ioctl logic for video capture in single and multi plane mode

vidioc_enum_fmt_vid_overlay pointer to the function that implements VIDIOC_ENUM_FMT ioctl logic for video overlay

vidioc_enum_fmt_vid_out pointer to the function that implements VIDIOC_ENUM_FMT ioctl logic for video output in single and multi plane mode

vidioc_enum_fmt_sdr_cap pointer to the function that implements VIDIOC_ENUM_FMT ioctl logic for Software Defined Radio capture

vidioc_enum_fmt_sdr_out pointer to the function that implements VIDIOC_ENUM_FMT ioctl logic for Software Defined Radio output

vidioc_enum_fmt_meta_cap pointer to the function that implements VIDIOC_ENUM_FMT ioctl logic for metadata capture

vidioc_enum_fmt_meta_out pointer to the function that implements VIDIOC_ENUM_FMT ioctl logic for metadata output

vidioc_g_fmt_vid_cap pointer to the function that implements VIDIOC_G_FMT ioctl logic for video capture in single plane mode

vidioc_g_fmt_vid_overlay pointer to the function that implements VIDIOC_G_FMT ioctl logic for video overlay

vidioc_g_fmt_vid_out pointer to the function that implements VIDIOC_G_FMT ioctl logic for video out in single plane mode

vidioc_g_fmt_vid_out_overlay pointer to the function that implements VIDIOC_G_FMT ioctl logic for video overlay output

vidioc_g_fmt_vbi_cap pointer to the function that implements VIDIOC_G_FMT ioctl logic for raw VBI capture

vidioc_g_fmt_vbi_out pointer to the function that implements VIDIOC_G_FMT ioctl logic for raw VBI output

vidioc_g_fmt_sliced_vbi_cap pointer to the function that implements VIDIOC_G_FMT ioctl logic for sliced VBI capture

vidioc_g_fmt_sliced_vbi_out pointer to the function that implements VIDIOC_G_FMT ioctl logic for sliced VBI output

vidioc_g_fmt_vid_cap_mplane pointer to the function that implements VIDIOC_G_FMT ioctl logic for video capture in multiple plane mode

vidioc_g_fmt_vid_out_mplane pointer to the function that implements VIDIOC_G_FMT ioctl logic for video out in multiplane plane mode

vidioc_g_fmt_sdr_cap pointer to the function that implements VIDIOC_G_FMT ioctl logic for Software Defined Radio capture

vidioc_g_fmt_sdr_out pointer to the function that implements VIDIOC_G_FMT ioctl logic for Software Defined Radio output

vidioc_g_fmt_meta_cap pointer to the function that implements VIDIOC_G_FMT ioctl logic for metadata capture

vidioc_g_fmt_meta_out pointer to the function that implements VIDIOC_G_FMT ioctl logic for metadata output

vidioc_s_fmt_vid_cap pointer to the function that implements VIDIOC_S_FMT ioctl logic for video capture in single plane mode

vidioc_s_fmt_vid_overlay pointer to the function that implements VIDIOC_S_FMT ioctl logic for video overlay

vidioc_s_fmt_vid_out pointer to the function that implements VIDIOC_S_FMT ioctl logic for video out in single plane mode

vidioc_s_fmt_vid_out_overlay pointer to the function that implements VIDIOC_S_FMT ioctl logic for video overlay output

vidioc_s_fmt_vbi_cap pointer to the function that implements VIDIOC_S_FMT ioctl logic for raw VBI capture

vidioc_s_fmt_vbi_out pointer to the function that implements VIDIOC_S_FMT ioctl logic for raw VBI output

vidioc_s_fmt_sliced_vbi_cap pointer to the function that implements VIDIOC_S_FMT ioctl logic for sliced VBI capture

vidioc_s_fmt_sliced_vbi_out pointer to the function that implements VIDIOC_S_FMT ioctl logic for sliced VBI output

vidioc_s_fmt_vid_cap_mplane pointer to the function that implements VIDIOC_S_FMT ioctl logic for video capture in multiple plane mode

vidioc_s_fmt_vid_out_mplane pointer to the function that implements VIDIOC_S_FMT ioctl logic for video out in multiplane plane mode

vidioc_s_fmt_sdr_cap pointer to the function that implements VIDIOC_S_FMT ioctl logic for Software Defined Radio capture

vidioc_s_fmt_sdr_out pointer to the function that implements VIDIOC_S_FMT ioctl logic for Software Defined Radio output

vidioc_s_fmt_meta_cap pointer to the function that implements VIDIOC_S_FMT ioctl logic for metadata capture

vidioc_s_fmt_meta_out pointer to the function that implements VIDIOC_S_FMT ioctl logic for metadata output

vidioc_try_fmt_vid_cap pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for video capture in single plane mode

vidioc_try_fmt_vid_overlay pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for video overlay

vidioc_try_fmt_vid_out pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for video out in single plane mode

vidioc_try_fmt_vid_out_overlay pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for video overlay output

vidioc_try_fmt_vbi_cap pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for raw VBI capture

vidioc_try_fmt_vbi_out pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for raw VBI output

vidioc_try_fmt_sliced_vbi_cap pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for sliced VBI capture

vidioc_try_fmt_sliced_vbi_out pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for sliced VBI output

vidioc_try_fmt_vid_cap_mplane pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for video capture in multiple plane mode

vidioc_try_fmt_vid_out_mplane pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for video out in multiplane plane mode

vidioc_try_fmt_sdr_cap pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for Software Defined Radio capture

vidioc_try_fmt_sdr_out pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for Software Defined Radio output

vidioc_try_fmt_meta_cap pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for metadata capture

vidioc_try_fmt_meta_out pointer to the function that implements VIDIOC_TRY_FMT ioctl logic for metadata output

vidioc_reqbufs pointer to the function that implements VIDIOC_REQBUFS ioctl

vidioc_querybuf pointer to the function that implements VIDIOC_QUERYBUF ioctl

vidioc_qbuf pointer to the function that implements VIDIOC_QBUF ioctl

vidioc_expbuf pointer to the function that implements VIDIOC_EXPBUF ioctl

vidioc_dqbuf pointer to the function that implements VIDIOC_DQBUF ioctl

vidioc_create_bufs pointer to the function that implements VIDIOC_CREATE_BUFS ioctl

vidioc_prepare_buf pointer to the function that implements VIDIOC_PREPARE_BUF ioctl

vidioc_overlay pointer to the function that implements VIDIOC_OVERLAY ioctl

vidioc_g_fbuf pointer to the function that implements VIDIOC_G_FBUF ioctl

vidioc_s_fbuf pointer to the function that implements VIDIOC_S_FBUF ioctl

vidioc_streamon pointer to the function that implements VIDIOC_STREAMON ioctl

vidioc_streamoff pointer to the function that implements VIDIOC_STREAMOFF ioctl

vidioc_g_std pointer to the function that implements VIDIOC_G_STD ioctl

vidioc_s_std pointer to the function that implements VIDIOC_S_STD ioctl

vidioc_querystd pointer to the function that implements VIDIOC_QUERYSTD ioctl

vidioc_enum_input pointer to the function that implements VIDIOC_ENUM_INPUT ioctl

vidioc_g_input pointer to the function that implements VIDIOC_G_INPUT ioctl

vidioc_s_input pointer to the function that implements VIDIOC_S_INPUT ioctl

vidioc_enum_output pointer to the function that implements VIDIOC_ENUM_OUTPUT ioctl

vidioc_g_output pointer to the function that implements VIDIOC_G_OUTPUT ioctl

vidioc_s_output pointer to the function that implements VIDIOC_S_OUTPUT ioctl

vidioc_queryctrl pointer to the function that implements VIDIOC_QUERYCTRL ioctl

vidioc_query_ext_ctrl pointer to the function that implements VIDIOC_QUERY_EXT_CTRL ioctl

vidioc_g_ctrl pointer to the function that implements VIDIOC_G_CTRL ioctl

vidioc_s_ctrl pointer to the function that implements VIDIOC_S_CTRL ioctl

vidioc_g_ext_ctrls pointer to the function that implements VIDIOC_G_EXT_CTRLS ioctl

vidioc_s_ext_ctrls pointer to the function that implements VIDIOC_S_EXT_CTRLS ioctl

vidioc_try_ext_ctrls pointer to the function that implements VIDIOC_TRY_EXT_CTRLS ioctl

vidioc_querymenu pointer to the function that implements VIDIOC_QUERYMENU ioctl

vidioc_enumaudio pointer to the function that implements VIDIOC_ENUMAUDIO ioctl

vidioc_g_audio pointer to the function that implements VIDIOC_G_AUDIO ioctl

vidioc_s_audio pointer to the function that implements VIDIOC_S_AUDIO ioctl

vidioc_enumaudout pointer to the function that implements VIDIOC_ENUMAUDOUT ioctl

vidioc_g_audout pointer to the function that implements VIDIOC_G_AUDOUT ioctl

vidioc_s_audout pointer to the function that implements VIDIOC_S_AUDOUT ioctl

vidioc_g_modulator pointer to the function that implements VIDIOC_G_MODULATOR ioctl

vidioc_s_modulator pointer to the function that implements VIDIOC_S_MODULATOR ioctl

vidioc_g_pixelaspect pointer to the function that implements the pixelaspect part of the VIDIOC_CROPCAP ioctl

vidioc_g_selection pointer to the function that implements VIDIOC_G_SELECTION ioctl

vidioc_s_selection pointer to the function that implements VIDIOC_S_SELECTION ioctl

vidioc_g_jpegcomp pointer to the function that implements VIDIOC_G_JPEGCOMP ioctl

vidioc_s_jpegcomp pointer to the function that implements VIDIOC_S_JPEGCOMP ioctl

vidioc_g_enc_index pointer to the function that implements VIDIOC_G_ENC_INDEX ioctl

vidioc_encoder_cmd pointer to the function that implements VIDIOC_ENCODER_CMD ioctl

vidioc_try_encoder_cmd pointer to the function that implements VIDIOC_TRY_ENCODER_CMD ioctl

vidioc_decoder_cmd pointer to the function that implements VIDIOC_DECODER_CMD ioctl

vidioc_try_decoder_cmd pointer to the function that implements VIDIOC_TRY_DECODER_CMD ioctl

vidioc_g_parm pointer to the function that implements VIDIOC_G_PARM ioctl

vidioc_s_parm pointer to the function that implements VIDIOC_S_PARM ioctl

vidioc_g_tuner pointer to the function that implements VIDIOC_G_TUNER ioctl

vidioc_s_tuner pointer to the function that implements VIDIOC_S_TUNER ioctl

vidioc_g_frequency pointer to the function that implements VIDIOC_G_FREQUENCY ioctl

vidioc_s_frequency pointer to the function that implements VIDIOC_S_FREQUENCY ioctl

vidioc_enum_freq_bands pointer to the function that implements VIDIOC_ENUM_FREQ_BANDS ioctl

vidioc_g_sliced_vbi_cap pointer to the function that implements VIDIOC_G_SLICED_VBI_CAP ioctl

vidioc_log_status pointer to the function that implements VIDIOC_LOG_STATUS ioctl

vidioc_s_hw_freq_seek pointer to the function that implements VIDIOC_S_HW_FREQ_SEEK ioctl

vidioc_g_register pointer to the function that implements VIDIOC_DBG_G_REGISTER ioctl

vidioc_s_register pointer to the function that implements VIDIOC_DBG_S_REGISTER ioctl

vidioc_g_chip_info pointer to the function that implements VIDIOC_DBG_G_CHIP_INFO ioctl

vidioc_enum_framesizes pointer to the function that implements VIDIOC_ENUM_FRAMESIZES ioctl

vidioc_enum_frameintervals pointer to the function that implements VIDIOC_ENUM_FRAMEINTERVALS ioctl

vidioc_s_dv_timings pointer to the function that implements VIDIOC_S_DV_TIMINGS ioctl

vidioc_g_dv_timings pointer to the function that implements VIDIOC_G_DV_TIMINGS ioctl

vidioc_query_dv_timings pointer to the function that implements VIDIOC_QUERY_DV_TIMINGS ioctl

vidioc_enum_dv_timings pointer to the function that implements VIDIOC_ENUM_DV_TIMINGS ioctl

vidioc_dv_timings_cap pointer to the function that implements VIDIOC_DV_TIMINGS_CAP ioctl

vidioc_g_edid pointer to the function that implements VIDIOC_G_EDID ioctl

vidioc_s_edid pointer to the function that implements VIDIOC_S_EDID ioctl

vidioc_subscribe_event pointer to the function that implements VIDIOC_SUBSCRIBE_EVENT ioctl

vidioc_unsubscribe_event pointer to the function that implements VIDIOC_UNSUBSCRIBE_EVENT ioctl

vidioc_default pointed used to allow other ioctls

const char * **v4l2_norm_to_name**(v4l2_std_id id)
Ancillary routine to analog TV standard name from its ID.

Parameters

v4l2_std_id id analog TV standard ID.

Return

returns a string with the name of the analog TV standard. If the standard is not found or if **id** points to multiple standard, it returns “Unknown”.

void **v4l2_video_std_frame_period**(int id, struct v4l2_fract * frameperiod)
Ancillary routine that fills a struct `v4l2_fract` pointer with the default framerate fraction.

Parameters

int id analog TV standard ID.

struct v4l2_fract * frameperiod struct `v4l2_fract` pointer to be filled

int **v4l2_video_std_construct**(struct v4l2_standard * vs, int id, const char * name)
Ancillary routine that fills in the fields of a `v4l2_standard` structure according to the **id** parameter.

Parameters

struct v4l2_standard * vs struct `v4l2_standard` pointer to be filled

int id analog TV standard ID.

const char * name name of the standard to be used

Description

Note: This ancillary routine is obsolete. Shouldn't be used on newer drivers.

---

int **v4l_video_std_enumstd**(struct v4l2_standard * vs, v4l2_std_id id)
Ancillary routine that fills in the fields of a `v4l2_standard` structure according to the **id** and **vs->index** parameters.

Parameters

struct v4l2_standard * vs struct `v4l2_standard` pointer to be filled.

v4l2_std_id id analog TV standard ID.

void **v4l_printk_ioctl**(const char * prefix, unsigned int cmd)
Ancillary routine that prints the ioctl in a human-readable format.

Parameters

const char * prefix prefix to be added at the ioctl prints.

unsigned int cmd ioctl name

Description

Note: If prefix != `NULL`, then it will issue a `printk(KERN_DEBUG "``s: “, prefix)``` first.

---

long int **v4l2_compat_ioctl32**(struct file * file, unsigned int cmd, unsigned long arg)
32 Bits compatibility layer for 64 bits processors

Parameters

struct file * file Pointer to struct `file`.

unsigned int cmd Ioctl name.

unsigned long arg Ioctl argument.

v4l2_kioctl
Typedef: Typedef used to pass an ioctl handler.

Syntax

```
long v4l2_kioctl (struct file * file, unsigned int cmd, void
* arg);
```

Parameters

struct file * file Pointer to struct `file`.

unsigned int cmd Ioctl name.

void * arg Ioctl argument.

long int **video_usercopy**(struct file * file, unsigned int cmd, unsigned long int arg, v4l2_kioctl func)
copies data from/to userspace memory when an ioctl is issued.

Parameters

struct file * file Pointer to struct `file`.

unsigned int cmd Ioctl name.

unsigned long int arg Ioctl argument.

v4l2_kioctl func function that will handle the ioctl

Description

Note: This routine should be used only inside the V4L2 core.

---

long int **video_ioctl2**(struct file * file, unsigned int cmd, unsigned long int arg)
Handles a V4L2 ioctl.

Parameters

struct file * file Pointer to struct `file`.

unsigned int cmd Ioctl name.

unsigned long int arg Ioctl argument.

Description

Method used to hancle an ioctl. Should be used to fill the `v4l2_ioctl_ops.unlocked_ioctl` on all V4L2 drivers.

53.1.27 Hauppauge TV EEPROM functions and data structures

enum **tveeprom_audio_processor**
Specifies the type of audio processor used on a Hauppauge device.

Constants

TVEEPROM_AUDPROC_NONE No audio processor present

TVEEPROM_AUDPROC_INTERNAL The audio processor is internal to the video processor

TVEEPROM_AUDPROC_MSP The audio processor is a MSPXXXX device

TVEEPROM_AUDPROC_OTHER The audio processor is another device

struct **tveeprom**
Contains the fields parsed from Hauppauge eeproms

Definition

```
struct tveeprom {
  u32 has_radio;
  u32 has_ir;
  u32 has_MAC_address;
  u32 tuner_type;
  u32 tuner_formats;
  u32 tuner_hauppauge_model;
  u32 tuner2_type;
  u32 tuner2_formats;
  u32 tuner2_hauppauge_model;
  u32 audio_processor;
  u32 decoder_processor;
  u32 model;
  u32 revision;
  u32 serial_number;
  char rev_str[5];
  u8 MAC_address[ETH_ALEN];
};
```

Members

has_radio 1 if the device has radio; 0 otherwise.

has_ir If has_ir == 0, then it is unknown what the IR capabilities are. Otherwise: bit 0) 1 (= IR capabilities are known); bit 1) IR receiver present; bit 2) IR transmitter (blaster) present.

has_MAC_address 0: no MAC, 1: MAC present, 2: unknown.

tuner_type type of the tuner (TUNER_*, as defined at include/media/tuner.h).

tuner_formats Supported analog TV standards (V4L2_STD_*).

tuner_hauppauge_model Hauppauge's code for the device model number.

tuner2_type type of the second tuner (TUNER_*, as defined at include/media/tuner.h).

tuner2_formats Tuner 2 supported analog TV standards (V4L2_STD_*).

tuner2_hauppauge_model tuner 2 Hauppauge's code for the device model number.

audio_processor analog audio decoder, as defined by enum tveeprom_audio_processor.

decoder_processor Hauppauge's code for the decoder chipset. Unused by the drivers, as they probe the decoder based on the PCI or USB ID.

model Hauppauge's model number

revision Card revision number

serial_number Card's serial number

rev_str Card revision converted to number

MAC_address MAC address for the network interface

void **tveeprom_hauppauge_analog**(struct tveeprom * tvee, unsigned char * eeprom_data)
Fill struct tveeprom using the contents of the eeprom previously filled at **eeprom_data** field.

Parameters

struct tveeprom * tvee Struct to where the eeprom parsed data will be filled;

unsigned char * eeprom_data Array with the contents of the eeprom_data. It should contain 256 bytes filled with the contents of the eeprom read from the Hauppauge device.

int **tveeprom_read**(struct i2c_client * c, unsigned char * eedata, int len)
Reads the contents of the eeprom found at the Hauppauge devices.

Parameters

struct i2c_client * c I2C client struct

unsigned char * eedata Array where the eeprom content will be stored.

int len Size of **eedata** array. If the eeprom content will be latter be parsed by `tveeprom_hauppauge_analog()`, len should be, at least, 256.

53.2 Digital TV (DVB) devices

Digital TV devices are implemented by several different drivers:

- A bridge driver that is responsible to talk with the bus where the other devices are connected (PCI, USB, SPI), bind to the other drivers and implement the digital demux logic (either in software or in hardware);
- Frontend drivers that are usually implemented as two separate drivers:
  - A tuner driver that implements the logic which commands the part of the hardware responsible for tuning into a digital TV transponder or physical channel. The output of a tuner is usually a baseband or Intermediate Frequency (IF) signal;
  - A demodulator driver (a.k.a “demod”) that implements the logic which commands the digital TV decoding hardware. The output of a demod is a digital stream, with multiple audio, video and data channels typically multiplexed using MPEG Transport Stream[1].

On most hardware, the frontend drivers talk with the bridge driver using an I2C bus.

[1] Some standards use TCP/IP for multiplexing data, like DVB-H (an abandoned standard, not used anymore) and ATSC version 3.0 current proposals. Currently, the DVB subsystem doesn’t implement those standards.

53.2.1 Digital TV Common functions

Math functions

Provide some commonly-used math functions, usually required in order to estimate signal strength and signal to noise measurements in dB.

unsigned int **intlog2**(u32 value)
computes log2 of a value; the result is shifted left by 24 bits

Parameters

u32 value The value (must be != 0)

Description

to use rational values you can use the following method:

intlog2(value) = intlog2(value * 2^x) - x * 2^24

Some usecase examples:

intlog2(8) will give 3 << 24 = 3 * 2^24

intlog2(9) will give 3 << 24 + ···= 3.16···* 2^24

intlog2(1.5) = intlog2(3) - 2^24 = 0.584···* 2^24

Return

log2(value) * 2^24

unsigned int **intlog10**(u32 value)
computes log10 of a value; the result is shifted left by 24 bits

Parameters

u32 value The value (must be != 0)

Description

to use rational values you can use the following method:

intlog10(value) = intlog10(value * 10^x) - x * 2^24

An usecase example:

intlog10(1000) will give 3 << 24 = 3 * 2^24

due to the implementation intlog10(1000) might be not exactly 3 * 2^24

look at intlog2 for similar examples

Return

log10(value) * 2^24

DVB devices

Those functions are responsible for handling the DVB device nodes.

enum **dvb_device_type**
type of the Digital TV device

Constants

DVB_DEVICE_SEC Digital TV standalone Common Interface (CI)

DVB_DEVICE_FRONTEND Digital TV frontend.

DVB_DEVICE_DEMUX Digital TV demux.

DVB_DEVICE_DVR Digital TV digital video record (DVR).

DVB_DEVICE_CA Digital TV Conditional Access (CA).

DVB_DEVICE_NET Digital TV network.

DVB_DEVICE_VIDEO Digital TV video decoder. Deprecated. Used only on av7110-av.

DVB_DEVICE_AUDIO Digital TV audio decoder. Deprecated. Used only on av7110-av.

DVB_DEVICE_OSD Digital TV On Screen Display (OSD). Deprecated. Used only on av7110.

struct **dvb_adapter**
represents a Digital TV adapter using Linux DVB API

Definition

```
struct dvb_adapter {
  int num;
  struct list_head list_head;
  struct list_head device_list;
  const char *name;
  u8 proposed_mac [6];
  void* priv;
  struct device *device;
  struct module *module;
  int mfe_shared;
  struct dvb_device *mfe_dvbdev;
  struct mutex mfe_lock;
#if defined(CONFIG_MEDIA_CONTROLLER_DVB);
  struct mutex mdev_lock;
  struct media_device *mdev;
  struct media_entity *conn;
  struct media_pad *conn_pads;
#endif;
};
```

Members

num Number of the adapter

list_head List with the DVB adapters

device_list List with the DVB devices

name Name of the adapter

proposed_mac proposed MAC address for the adapter

priv private data

device pointer to struct device

module pointer to struct module

mfe_shared indicates mutually exclusive frontends. Use of this flag is currently deprecated.

mfe_dvbdev Frontend device in use, in the case of MFE

mfe_lock Lock to prevent using the other frontends when MFE is used.

mdev_lock Protect access to the mdev pointer.

mdev pointer to struct media_device, used when the media controller is used.

conn RF connector. Used only if the device has no separate tuner.

conn_pads pointer to struct media_pad associated with **conn**;

struct **dvb_device**
represents a DVB device node

Definition

```
struct dvb_device {
  struct list_head list_head;
  const struct file_operations *fops;
  struct dvb_adapter *adapter;
  enum dvb_device_type type;
  int minor;
  u32 id;
  int readers;
  int writers;
  int users;
  wait_queue_head_t wait_queue;
  int (*kernel_ioctl)(struct file *file, unsigned int cmd, void *arg);
#if defined(CONFIG_MEDIA_CONTROLLER_DVB);
  const char *name;
  struct media_intf_devnode *intf_devnode;
  unsigned tsout_num_entities;
  struct media_entity *entity, *tsout_entity;
  struct media_pad *pads, *tsout_pads;
#endif;
  void *priv;
};
```

Members

list_head List head with all DVB devices

fops pointer to struct file_operations

adapter pointer to the adapter that holds this device node

type type of the device, as defined by `enum dvb_device_type`.

minor devnode minor number. Major number is always DVB_MAJOR.

id device ID number, inside the adapter

readers Initialized by the caller. Each call to open() in Read Only mode decreases this counter by one.

writers Initialized by the caller. Each call to open() in Read/Write mode decreases this counter by one.

users Initialized by the caller. Each call to open() in any mode decreases this counter by one.

wait_queue wait queue, used to wait for certain events inside one of the DVB API callers

kernel_ioctl callback function used to handle ioctl calls from userspace.

name Name to be used for the device at the Media Controller

intf_devnode Pointer to media_intf_devnode. Used by the dvbdev core to store the MC device node interface

tsout_num_entities Number of Transport Stream output entities

entity pointer to struct media_entity associated with the device node

tsout_entity array with MC entities associated to each TS output node

pads pointer to struct media_pad associated with **entity**;

tsout_pads array with the source pads for each **tsout_entity**

priv private data

Description

This structure is used by the DVB core (frontend, CA, net, demux) in order to create the device nodes. Usually, driver should not initialize this struct diretly.

int **dvb_register_adapter**(struct dvb_adapter * adap, const char * name, struct module * module, struct device * device, short * adapter_nums)
Registers a new DVB adapter

Parameters

struct dvb_adapter * adap pointer to struct dvb_adapter

const char * name Adapter's name

struct module * module initialized with THIS_MODULE at the caller

struct device * device pointer to struct device that corresponds to the device driver

short * adapter_nums Array with a list of the numbers for **dvb_register_adapter**; to select among them. Typically, initialized with: DVB_DEFINE_MOD_OPT_ADAPTER_NR(adapter_nums)

int **dvb_unregister_adapter**(struct dvb_adapter * adap)
Unregisters a DVB adapter

Parameters

struct dvb_adapter * adap pointer to struct dvb_adapter

int **dvb_register_device**(struct dvb_adapter * adap, struct dvb_device ** pdvbdev, const struct dvb_device * template, void * priv, enum dvb_device_type type, int demux_sink_pads)
Registers a new DVB device

Parameters

struct dvb_adapter * adap pointer to struct dvb_adapter

struct dvb_device ** pdvbdev pointer to the place where the new struct dvb_device will be stored

const struct dvb_device * template Template used to create `pdvbdev`;

void * priv private data

enum dvb_device_type type type of the device, as defined by `enum dvb_device_type`.

int demux_sink_pads Number of demux outputs, to be used to create the TS outputs via the Media Controller.

void **dvb_remove_device**(struct dvb_device * dvbdev)
Remove a registered DVB device

Parameters

struct dvb_device * dvbdev pointer to struct dvb_device

Description

This does not free memory. To do that, call `dvb_free_device()`.

void **dvb_free_device**(struct dvb_device * dvbdev)
Free memory occupied by a DVB device.

Parameters

struct dvb_device * dvbdev pointer to struct dvb_device

Description

Call `dvb_unregister_device()` before calling this function.

void **dvb_unregister_device**(struct dvb_device * dvbdev)
Unregisters a DVB device

Parameters

struct dvb_device * dvbdev pointer to struct dvb_device

Description

This is a combination of `dvb_remove_device()` and `dvb_free_device()`. Using this function is usually a mistake, and is often an indicator for a use-after-free bug (when a userspace process keeps a file handle to a detached device).

int **dvb_create_media_graph**(struct dvb_adapter * adap, bool create_rf_connector)
Creates media graph for the Digital TV part of the device.

Parameters

struct dvb_adapter * adap pointer to `struct dvb_adapter`

bool create_rf_connector if true, it creates the RF connector too

Description

This function checks all DVB-related functions at the media controller entities and creates the needed links for the media graph. It is capable of working with multiple tuners or multiple frontends, but it won't create links if the device has multiple tuners and multiple frontends or if the device has multiple muxes. In such case, the caller driver should manually create the remaining links.

void **dvb_register_media_controller**(struct dvb_adapter * adap, struct media_device * mdev)
registers a media controller at DVB adapter

Parameters

struct dvb_adapter * adap pointer to `struct dvb_adapter`

struct media_device * mdev pointer to `struct media_device`

struct media_device * **dvb_get_media_controller**(struct dvb_adapter * adap)
gets the associated media controller

Parameters

struct dvb_adapter * adap pointer to `struct dvb_adapter`

int **dvb_generic_open**(struct inode * inode, struct file * file)
Digital TV open function, used by DVB devices

Parameters

struct inode * inode pointer to `struct inode`.

struct file * file pointer to `struct file`.

Description

Checks if a DVB devnode is still valid, and if the permissions are OK and increment negative use count.

int **dvb_generic_release**(struct inode * inode, struct file * file)
Digital TV close function, used by DVB devices

Parameters

struct inode * inode pointer to `struct inode`.

struct file * file pointer to `struct file`.

Description

Checks if a DVB devnode is still valid, and if the permissions are OK and decrement negative use count.

long **dvb_generic_ioctl**(struct file * file, unsigned int cmd, unsigned long arg)
Digital TV close function, used by DVB devices

Parameters

struct file * file pointer to `struct file`.

unsigned int cmd Ioctl name.

unsigned long arg Ioctl argument.

Description

Checks if a DVB devnode and struct dvbdev.kernel_ioctl is still valid. If so, calls `dvb_usercopy()`.

int **dvb_usercopy**(struct file * file, unsigned int cmd, unsigned long arg, int (*func)(struct file *file, unsigned int cmd, void *arg))
copies data from/to userspace memory when an ioctl is issued.

Parameters

struct file * file Pointer to struct `file`.

unsigned int cmd Ioctl name.

unsigned long arg Ioctl argument.

int (*)(struct file *file, unsigned int cmd, void *arg) func function that will actually handle the ioctl

Description

Ancillary function that uses ioctl direction and size to copy from userspace. Then, it calls **func**, and, if needed, data is copied back to userspace.

struct i2c_client * **dvb_module_probe**(const char * module_name, const char * name, struct i2c_adapter * adap, unsigned char addr, void * platform_data)
helper routine to probe an I2C module

Parameters

const char * module_name Name of the I2C module to be probed

const char * name Optional name for the I2C module. Used for debug purposes. If `NULL`, defaults to **module_name**.

struct i2c_adapter * adap pointer to `struct i2c_adapter` that describes the I2C adapter where the module will be bound.

unsigned char addr I2C address of the adapter, in 7-bit notation.

void * platform_data Platform data to be passed to the I2C module probed.

Description

This function binds an I2C device into the DVB core. Should be used by all drivers that use I2C bus to control the hardware. A module bound with `dvb_module_probe()` should use `dvb_module_release()` to unbind.

Note: In the past, DVB modules (mainly, frontends) were bound via `dvb_attach()` macro, with does an ugly hack, using I2C low level functions. Such usage is deprecated and will be removed soon. Instead, use this routine.

Return

On success, return an `struct i2c_client`, pointing the the bound I2C device. `NULL` otherwise.

void **dvb_module_release**(struct i2c_client * client)
releases an I2C device allocated with `dvb_module_probe()`.

Parameters

`struct i2c_client * client` pointer to `struct i2c_client` with the I2C client to be released. can be `NULL`.

Description

This function should be used to free all resources reserved by `dvb_module_probe()` and unbinding the I2C hardware.

dvb_attach(FUNCTION, ARGS)
attaches a DVB frontend into the DVB core.

Parameters

`FUNCTION` function on a frontend module to be called.

`ARGS` **FUNCTION** arguments.

Description

This ancillary function loads a frontend module in runtime and runs the **FUNCTION** function there, with **ARGS**. As it increments symbol usage cont, at unregister, `dvb_detach()` should be called.

Note: In the past, DVB modules (mainly, frontends) were bound via `dvb_attach()` macro, with does an ugly hack, using I2C low level functions. Such usage is deprecated and will be removed soon. Instead, you should use `dvb_module_probe()`.

---

dvb_detach(FUNC)
detaches a DVB frontend loaded via `dvb_attach()`

Parameters

`FUNC` attach function

Description

Decrements usage count for a function previously called via `dvb_attach()`.

Digital TV Ring buffer

Those routines implement ring buffers used to handle digital TV data and copy it from/to userspace.

Note:

1) For performance reasons read and write routines don't check buffer sizes and/or number of bytes free/available. This has to be done before these routines are called. For example:

```
/* write @buflen: bytes */
free = dvb_ringbuffer_free(rbuf);
if (free >= buflen)
        count = dvb_ringbuffer_write(rbuf, buffer, buflen);
else
        /* do something */

/* read min. 1000, max. @bufsize: bytes */
avail = dvb_ringbuffer_avail(rbuf);
if (avail >= 1000)
        count = dvb_ringbuffer_read(rbuf, buffer, min(avail,␣
↪bufsize));
else
        /* do something */
```

2) If there is exactly one reader and one writer, there is no need to lock read or write operations. Two or more readers must be locked against each other. Flushing the buffer counts as a read operation. Resetting the buffer counts as a read and write operation. Two or more writers must be locked against each other.

---

struct **dvb_ringbuffer**
Describes a ring buffer used at DVB framework

Definition

```
struct dvb_ringbuffer {
  u8 *data;
  ssize_t size;
  ssize_t pread;
  ssize_t pwrite;
  int error;
  wait_queue_head_t queue;
  spinlock_t lock;
};
```

Members

data Area were the ringbuffer data is written

size size of the ringbuffer

pread next position to read

pwrite next position to write

error used by ringbuffer clients to indicate that an error happened.

queue Wait queue used by ringbuffer clients to indicate when buffer was filled

lock Spinlock used to protect the ringbuffer

void **dvb_ringbuffer_init**(struct dvb_ringbuffer * rbuf, void * data, size_t len)
initialize ring buffer, lock and queue

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

void * data pointer to the buffer where the data will be stored

size_t len bytes from ring buffer into **buf**

int **dvb_ringbuffer_empty**(struct dvb_ringbuffer * rbuf)
test whether buffer is empty

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

ssize_t **dvb_ringbuffer_free**(struct dvb_ringbuffer * rbuf)
returns the number of free bytes in the buffer

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

Return

number of free bytes in the buffer

ssize_t **dvb_ringbuffer_avail**(struct dvb_ringbuffer * rbuf)
returns the number of bytes waiting in the buffer

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

Return

number of bytes waiting in the buffer

void **dvb_ringbuffer_reset**(struct dvb_ringbuffer * rbuf)
resets the ringbuffer to initial state

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

Description

Resets the read and write pointers to zero and flush the buffer.

This counts as a read and write operation

void **dvb_ringbuffer_flush**(struct dvb_ringbuffer * rbuf)
flush buffer

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

void **dvb_ringbuffer_flush_spinlock_wakeup**(struct dvb_ringbuffer * rbuf)
flush buffer protected by spinlock and wake-up waiting task(s)

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

DVB_RINGBUFFER_PEEK(rbuf, offs)
peek at byte **offs** in the buffer

Parameters

rbuf pointer to struct dvb_ringbuffer

offs offset inside the ringbuffer

DVB_RINGBUFFER_SKIP(rbuf, num)
advance read ptr by **num** bytes

Parameters

rbuf pointer to struct dvb_ringbuffer

num number of bytes to advance

ssize_t **dvb_ringbuffer_read_user**(struct dvb_ringbuffer * rbuf, u8 __user * buf, size_t len)
Reads a buffer into a user pointer

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

u8 __user * buf pointer to the buffer where the data will be stored

size_t len bytes from ring buffer into **buf**

Description

This variant assumes that the buffer is a memory at the userspace. So, it will internally call copy_to_user().

Return

number of bytes transferred or -EFAULT

void **dvb_ringbuffer_read**(struct dvb_ringbuffer * rbuf, u8 * buf, size_t len)
Reads a buffer into a pointer

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

u8 * buf pointer to the buffer where the data will be stored

size_t len bytes from ring buffer into **buf**

Description

This variant assumes that the buffer is a memory at the Kernel space

Return

number of bytes transferred or -EFAULT

DVB_RINGBUFFER_WRITE_BYTE(rbuf, byte)
write single byte to ring buffer

Parameters

rbuf pointer to struct dvb_ringbuffer

byte byte to write

ssize_t **dvb_ringbuffer_write**(struct dvb_ringbuffer * rbuf, const u8 * buf, size_t len)
Writes a buffer into the ringbuffer

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

const u8 * buf pointer to the buffer where the data will be read

size_t len bytes from ring buffer into **buf**

Description

This variant assumes that the buffer is a memory at the Kernel space

Return

number of bytes transferred or -EFAULT

ssize_t **dvb_ringbuffer_write_user**(struct dvb_ringbuffer * rbuf, const u8 __user * buf, size_t len)
Writes a buffer received via a user pointer

Parameters

struct dvb_ringbuffer * rbuf pointer to struct dvb_ringbuffer

const u8 __user * buf pointer to the buffer where the data will be read

size_t len bytes from ring buffer into **buf**

Description

This variant assumes that the buffer is a memory at the userspace. So, it will internally call copy_from_user().

Return

number of bytes transferred or -EFAULT

ssize_t **dvb_ringbuffer_pkt_write**(struct dvb_ringbuffer * rbuf, u8 * buf, size_t len)
Write a packet into the ringbuffer.

Parameters

struct dvb_ringbuffer * rbuf Ringbuffer to write to.

u8 * buf Buffer to write.

size_t len Length of buffer (currently limited to 65535 bytes max).

Return

Number of bytes written, or -EFAULT, -ENOMEM, -EVINAL.

ssize_t **dvb_ringbuffer_pkt_read_user**(struct dvb_ringbuffer * rbuf, size_t idx, int offset, u8 __user * buf, size_t len)
Read from a packet in the ringbuffer.

Parameters

struct dvb_ringbuffer * rbuf Ringbuffer concerned.

size_t idx Packet index as returned by `dvb_ringbuffer_pkt_next()`.

int offset Offset into packet to read from.

u8 __user * buf Destination buffer for data.

size_t len Size of destination buffer.

Return

Number of bytes read, or -EFAULT.

Description

Note: unlike `dvb_ringbuffer_read()`, this does **NOT** update the read pointer in the ringbuffer. You must use `dvb_ringbuffer_pkt_dispose()` to mark a packet as no longer required.

---

ssize_t **dvb_ringbuffer_pkt_read**(struct dvb_ringbuffer * rbuf, size_t idx, int offset, u8 * buf, size_t len)
Read from a packet in the ringbuffer.

Parameters

struct dvb_ringbuffer * rbuf Ringbuffer concerned.

size_t idx Packet index as returned by `dvb_ringbuffer_pkt_next()`.

int offset Offset into packet to read from.

u8 * buf Destination buffer for data.

size_t len Size of destination buffer.

Note

unlike `dvb_ringbuffer_read_user()`, this DOES update the read pointer in the ringbuffer.

Return

Number of bytes read, or -EFAULT.

void **dvb_ringbuffer_pkt_dispose**(struct dvb_ringbuffer * rbuf, size_t idx)
Dispose of a packet in the ring buffer.

Parameters

struct dvb_ringbuffer * rbuf Ring buffer concerned.

size_t idx Packet index as returned by `dvb_ringbuffer_pkt_next()`.

ssize_t **dvb_ringbuffer_pkt_next**(struct dvb_ringbuffer * rbuf, size_t idx, size_t * pktlen)
Get the index of the next packet in a ringbuffer.

Parameters

struct dvb_ringbuffer * rbuf Ringbuffer concerned.

size_t idx Previous packet index, or -1 to return the first packet index.

size_t * pktlen On success, will be updated to contain the length of the packet in bytes. returns Packet index (if >=0), or -1 if no packets available.

Digital TV VB2 handler

enum **dvb_buf_type**
types of Digital TV memory-mapped buffers

Constants

DVB_BUF_TYPE_CAPTURE buffer is filled by the Kernel, with a received Digital TV stream

enum **dvb_vb2_states**
states to control VB2 state machine

Constants

DVB_VB2_STATE_NONE VB2 engine not initialized yet, init failed or VB2 was released.

DVB_VB2_STATE_INIT VB2 engine initialized.

DVB_VB2_STATE_REQBUFS Buffers were requested

DVB_VB2_STATE_STREAMON VB2 is streaming. Callers should not check it directly. Instead, they should use `dvb_vb2_is_streaming()`.

Note

Description

Callers should not touch at the state machine directly. This is handled inside dvb_vb2.c.

struct **dvb_buffer**
video buffer information for v4l2.

Definition

```
struct dvb_buffer {
  struct vb2_buffer       vb;
  struct list_head        list;
};
```

Members

vb embedded struct `vb2_buffer`.

list list of `struct dvb_buffer`.

struct **dvb_vb2_ctx**
control struct for VB2 handler

Definition

```
struct dvb_vb2_ctx {
  struct vb2_queue        vb_q;
  struct mutex            mutex;
  spinlock_t slock;
  struct list_head        dvb_q;
  struct dvb_buffer       *buf;
  int offset;
  int remain;
  int state;
  int buf_siz;
  int buf_cnt;
  int nonblocking;
  enum dmx_buffer_flags flags;
  u32 count;
  char name[DVB_VB2_NAME_MAX + 1];
};
```

Members

vb_q pointer to `struct vb2_queue` with videobuf2 queue.

mutex mutex to serialize vb2 operations. Used by vb2 core `wait_prepare` and `wait_finish` operations.

slock spin lock used to protect buffer filling at dvb_vb2.c.

dvb_q List of buffers that are not filled yet.

buf Pointer to the buffer that are currently being filled.

offset index to the next position at the **buf** to be filled.

remain How many bytes are left to be filled at **buf**.

state bitmask of buffer states as defined by `enum dvb_vb2_states`.

buf_siz size of each VB2 buffer.

buf_cnt number of VB2 buffers.

nonblocking If different than zero, device is operating on non-blocking mode.

flags buffer flags as defined by `enum dmx_buffer_flags`. Filled only at `DMX_DQBUF`. `DMX_QBUF` should zero this field.

count monotonic counter for filled buffers. Helps to identify data stream loses. Filled only at `DMX_DQBUF`. `DMX_QBUF` should zero this field.

name name of the device type. Currently, it can either be “dvr” or “demux_filter” .

int **dvb_vb2_init**(struct dvb_vb2_ctx * ctx, const char * name, int non_blocking)
initializes VB2 handler

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

const char * name name for the VB2 handler

int non_blocking if not zero, it means that the device is at non-blocking mode

int **dvb_vb2_release**(struct dvb_vb2_ctx * ctx)
Releases the VB2 handler allocated resources and put **ctx** at DVB_VB2_STATE_NONE state.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

int **dvb_vb2_is_streaming**(struct dvb_vb2_ctx * ctx)
checks if the VB2 handler is streaming

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

Return

0 if not streaming, 1 otherwise.

int **dvb_vb2_fill_buffer**(struct dvb_vb2_ctx * ctx, const unsigned char * src, int len, enum dmx_buffer_flags * buffer_flags)
fills a VB2 buffer

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

const unsigned char * src place where the data is stored

int len number of bytes to be copied from **src**

enum dmx_buffer_flags * buffer_flags pointer to buffer flags as defined by `enum dmx_buffer_flags`. can be NULL.

__poll_t **dvb_vb2_poll**(struct dvb_vb2_ctx * ctx, struct file * file, poll_table * wait)
Wrapper to `vb2_core_streamon()` for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

struct file * file `struct file` argument passed to the poll file operation handler.

poll_table * wait `poll_table` wait argument passed to the poll file operation handler.

Description

Implements poll syscall() logic.

int **dvb_vb2_stream_on**(struct dvb_vb2_ctx * ctx)
Wrapper to `vb2_core_streamon()` for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

Description

Starts dvb streaming

int **dvb_vb2_stream_off**(struct dvb_vb2_ctx * ctx)
Wrapper to `vb2_core_streamoff()` for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

Description

Stops dvb streaming

int **dvb_vb2_reqbufs**(struct dvb_vb2_ctx * ctx, struct dmx_requestbuffers * req)
Wrapper to `vb2_core_reqbufs()` for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

struct dmx_requestbuffers * req `struct dmx_requestbuffers` passed from userspace in order to handle `DMX_REQBUFS`.

Description

Initiate streaming by requesting a number of buffers. Also used to free previously requested buffers, is `req->count` is zero.

int **dvb_vb2_querybuf**(struct dvb_vb2_ctx * ctx, struct dmx_buffer * b)
Wrapper to `vb2_core_querybuf()` for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

struct dmx_buffer * b `struct dmx_buffer` passed from userspace in order to handle `DMX_QUERYBUF`.

int **dvb_vb2_expbuf**(struct dvb_vb2_ctx * ctx, struct dmx_exportbuffer * exp)
Wrapper to `vb2_core_expbuf()` for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

struct dmx_exportbuffer * exp `struct dmx_exportbuffer` passed from userspace in order to handle `DMX_EXPBUF`.

Description

Export a buffer as a file descriptor.

int **dvb_vb2_qbuf**(struct dvb_vb2_ctx * ctx, struct dmx_buffer * b)
Wrapper to `vb2_core_qbuf()` for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

struct dmx_buffer * b struct dmx_buffer passed from userspace in order to handle DMX_QBUF.

Description

Queue a Digital TV buffer as requested by userspace

int **dvb_vb2_dqbuf**(struct dvb_vb2_ctx * ctx, struct dmx_buffer * b)
Wrapper to vb2_core_dqbuf() for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

struct dmx_buffer * b struct dmx_buffer passed from userspace in order to handle DMX_DQBUF.

Description

Dequeue a Digital TV buffer to the userspace

int **dvb_vb2_mmap**(struct dvb_vb2_ctx * ctx, struct vm_area_struct * vma)
Wrapper to vb2_mmap() for Digital TV buffer handling.

Parameters

struct dvb_vb2_ctx * ctx control struct for VB2 handler

struct vm_area_struct * vma pointer to struct vm_area_struct with the vma passed to the mmap file operation handler in the driver.

Description

map Digital TV video buffers into application address space.

53.2.2 Digital TV Frontend kABI

Digital TV Frontend

The Digital TV Frontend kABI defines a driver-internal interface for registering low-level, hardware specific driver to a hardware independent frontend layer. It is only of interest for Digital TV device driver writers. The header file for this API is named dvb_frontend.h and located in include/media/.

Demodulator driver

The demodulator driver is responsible for talking with the decoding part of the hardware. Such driver should implement dvb_frontend_ops, which tells what type of digital TV standards are supported, and points to a series of functions that allow the DVB core to command the hardware via the code under include/media/dvb_frontend.c.

A typical example of such struct in a driver foo is:

```
static struct dvb_frontend_ops foo_ops = {
        .delsys = { SYS_DVBT, SYS_DVBT2, SYS_DVBC_ANNEX_A },
        .info = {
                .name   = "foo DVB-T/T2/C driver",
                .caps = FE_CAN_FEC_1_2 |
                        FE_CAN_FEC_2_3 |
                        FE_CAN_FEC_3_4 |
                        FE_CAN_FEC_5_6 |
                        FE_CAN_FEC_7_8 |
                        FE_CAN_FEC_AUTO |
                        FE_CAN_QPSK |
                        FE_CAN_QAM_16 |
                        FE_CAN_QAM_32 |
                        FE_CAN_QAM_64 |
                        FE_CAN_QAM_128 |
                        FE_CAN_QAM_256 |
                        FE_CAN_QAM_AUTO |
                        FE_CAN_TRANSMISSION_MODE_AUTO |
                        FE_CAN_GUARD_INTERVAL_AUTO |
                        FE_CAN_HIERARCHY_AUTO |
                        FE_CAN_MUTE_TS |
                        FE_CAN_2G_MODULATION,
                .frequency_min = 42000000, /* Hz */
                .frequency_max = 1002000000, /* Hz */
                .symbol_rate_min = 870000,
                .symbol_rate_max = 11700000
        },
        .init = foo_init,
        .sleep = foo_sleep,
        .release = foo_release,
        .set_frontend = foo_set_frontend,
        .get_frontend = foo_get_frontend,
        .read_status = foo_get_status_and_stats,
        .tune = foo_tune,
        .i2c_gate_ctrl = foo_i2c_gate_ctrl,
        .get_frontend_algo = foo_get_algo,
};
```

A typical example of such struct in a driver `bar` meant to be used on Satellite TV reception is:

```
static const struct dvb_frontend_ops bar_ops = {
        .delsys = { SYS_DVBS, SYS_DVBS2 },
        .info = {
                .name           = "Bar DVB-S/S2 demodulator",
                .frequency_min  = 500000, /* KHz */
                .frequency_max  = 2500000, /* KHz */
                .frequency_stepsize     = 0,
                .symbol_rate_min = 1000000,
                .symbol_rate_max = 45000000,
                .symbol_rate_tolerance = 500,
                .caps = FE_CAN_INVERSION_AUTO |
                        FE_CAN_FEC_AUTO |
                        FE_CAN_QPSK,
        },
        .init = bar_init,
```

(continues on next page)

(continued from previous page)

```
        .sleep = bar_sleep,
        .release = bar_release,
        .set_frontend = bar_set_frontend,
        .get_frontend = bar_get_frontend,
        .read_status = bar_get_status_and_stats,
        .i2c_gate_ctrl = bar_i2c_gate_ctrl,
        .get_frontend_algo = bar_get_algo,
        .tune = bar_tune,

        /* Satellite-specific */
        .diseqc_send_master_cmd = bar_send_diseqc_msg,
        .diseqc_send_burst = bar_send_burst,
        .set_tone = bar_set_tone,
        .set_voltage = bar_set_voltage,
};
```

Note:

1) For satellite digital TV standards (DVB-S, DVB-S2, ISDB-S), the frequencies are specified in kHz, while, for terrestrial and cable standards, they' re specified in Hz. Due to that, if the same frontend supports both types, you' ll need to have two separate `dvb_frontend_ops` structures, one for each standard.

2) The `.i2c_gate_ctrl` field is present only when the hardware has allows controlling an I2C gate (either directly of via some GPIO pin), in order to remove the tuner from the I2C bus after a channel is tuned.

3) All new drivers should implement the DVBv5 statistics via `.read_status`. Yet, there are a number of callbacks meant to get statistics for signal strength, S/N and UCB. Those are there to provide backward compatibility with legacy applications that don' t support the DVBv5 API. Implementing those callbacks are optional. Those callbacks may be removed in the future, after we have all existing drivers supporting DVBv5 stats.

4) Other callbacks are required for satellite TV standards, in order to control LNBf and DiSEqC: `.diseqc_send_master_cmd`, `.diseqc_send_burst`, `.set_tone`, `.set_voltage`.

The `include/media/dvb_frontend.c` has a kernel thread which is responsible for tuning the device. It supports multiple algorithms to detect a channel, as defined at enum `dvbfe_algo()`.

The algorithm to be used is obtained via `.get_frontend_algo`. If the driver doesn't fill its field at struct `dvb_frontend_ops`, it will default to `DVBFE_ALGO_SW`, meaning that the dvb-core will do a zigzag when tuning, e. g. it will try first to use the specified center frequency `f`, then, it will do `f` + Δ, `f` - Δ, `f` + 2 x Δ, `f` - 2 x Δ and so on.

If the hardware has internally a some sort of zigzag algorithm, you should define a `.get_frontend_algo` function that would return `DVBFE_ALGO_HW`.

Note: The core frontend support also supports a third type (`DVBFE_ALGO_CUSTOM`), in order to allow the driver to define its own hardware-assisted algorithm. Very few

hardware need to use it nowadays. Using DVBFE_ALGO_CUSTOM require to provide other function callbacks at struct dvb_frontend_ops.

Attaching frontend driver to the bridge driver

Before using the Digital TV frontend core, the bridge driver should attach the frontend demod, tuner and SEC devices and call dvb_register_frontend(), in order to register the new frontend at the subsystem. At device detach/removal, the bridge driver should call dvb_unregister_frontend() to remove the frontend from the core and then dvb_frontend_detach() to free the memory allocated by the frontend drivers.

The drivers should also call dvb_frontend_suspend() as part of their handler for the device_driver.suspend(), and dvb_frontend_resume() as part of their handler for device_driver.resume().

A few other optional functions are provided to handle some special cases.

Digital TV Frontend statistics

Introduction

Digital TV frontends provide a range of statistics meant to help tuning the device and measuring the quality of service.

For each statistics measurement, the driver should set the type of scale used, or FE_SCALE_NOT_AVAILABLE if the statistics is not available on a given time. Drivers should also provide the number of statistics for each type. that's usually 1 for most video standards[1].

Drivers should initialize each statistic counters with length and scale at its init code. For example, if the frontend provides signal strength, it should have, on its init code:

```
struct dtv_frontend_properties *c = &state->fe.dtv_property_cache;

c->strength.len = 1;
c->strength.stat[0].scale = FE_SCALE_NOT_AVAILABLE;
```

And, when the statistics got updated, set the scale:

```
c->strength.stat[0].scale = FE_SCALE_DECIBEL;
c->strength.stat[0].uvalue = strength;
```

[1] For ISDB-T, it may provide both a global statistics and a per-layer set of statistics. On such cases, len should be equal to 4. The first value corresponds to the global stat; the other ones to each layer, e. g.:

- c->cnr.stat[0] for global S/N carrier ratio,
- c->cnr.stat[1] for Layer A S/N carrier ratio,
- c->cnr.stat[2] for layer B S/N carrier ratio,
- c->cnr.stat[3] for layer C S/N carrier ratio.

Note: Please prefer to use FE_SCALE_DECIBEL instead of FE_SCALE_RELATIVE for signal strength and CNR measurements.

Groups of statistics

There are several groups of statistics currently supported:

Signal strength (DTV-STAT-SIGNAL-STRENGTH)

- Measures the signal strength level at the analog part of the tuner or demod.
- Typically obtained from the gain applied to the tuner and/or frontend in order to detect the carrier. When no carrier is detected, the gain is at the maximum value (so, strength is on its minimal).
- As the gain is visible through the set of registers that adjust the gain, typically, this statistics is always available[2].
- Drivers should try to make it available all the times, as these statistics can be used when adjusting an antenna position and to check for troubles at the cabling.

Carrier Signal to Noise ratio (DTV-STAT-CNR)

- Signal to Noise ratio for the main carrier.
- Signal to Noise measurement depends on the device. On some hardware, it is available when the main carrier is detected. On those hardware, CNR measurement usually comes from the tuner (e. g. after FE_HAS_CARRIER, see fe_status).

  On other devices, it requires inner FEC decoding, as the frontend measures it indirectly from other parameters (e. g. after FE_HAS_VITERBI, see fe_status).

  Having it available after inner FEC is more common.

Bit counts post-FEC (DTV-STAT-POST-ERROR-BIT-COUNT and DTV-STAT-POST-TOTAL

- Those counters measure the number of bits and bit errors errors after the forward error correction (FEC) on the inner coding block (after Viterbi, LDPC or other inner code).
- Due to its nature, those statistics depend on full coding lock (e. g. after FE_HAS_SYNC or after FE_HAS_LOCK, see fe_status).

Bit counts pre-FEC (DTV-STAT-PRE-ERROR-BIT-COUNT and DTV-STAT-PRE-TOTAL-BI

[2] On a few devices, the gain keeps floating if there is no carrier. On such devices, strength report should check first if carrier is detected at the tuner (FE_HAS_CARRIER, see fe_status), and otherwise return the lowest possible value.

- Those counters measure the number of bits and bit errors errors before the forward error correction (FEC) on the inner coding block (before Viterbi, LDPC or other inner code).
- Not all frontends provide this kind of statistics.
- Due to its nature, those statistics depend on inner coding lock (e. g. after FE_HAS_VITERBI, see fe_status).

Block counts (DTV-STAT-ERROR-BLOCK-COUNT and DTV-STAT-TOTAL-BLOCK-COUN

- Those counters measure the number of blocks and block errors errors after the forward error correction (FEC) on the inner coding block (before Viterbi, LDPC or other inner code).
- Due to its nature, those statistics depend on full coding lock (e. g. after FE_HAS_SYNC or after FE_HAS_LOCK, see fe_status).

Note: All counters should be monotonically increased as they' re collected from the hardware.

A typical example of the logic that handle status and statistics is:

```
static int foo_get_status_and_stats(struct dvb_frontend *fe)
{
        struct foo_state *state = fe->demodulator_priv;
        struct dtv_frontend_properties *c = &fe->dtv_property_cache;

        int rc;
        enum fe_status *status;

        /* Both status and strength are always available */
        rc = foo_read_status(fe, &status);
        if (rc < 0)
                return rc;

        rc = foo_read_strength(fe);
        if (rc < 0)
                return rc;

        /* Check if CNR is available */
        if (!(fe->status & FE_HAS_CARRIER))
                return 0;

        rc = foo_read_cnr(fe);
        if (rc < 0)
                return rc;

        /* Check if pre-BER stats are available */
        if (!(fe->status & FE_HAS_VITERBI))
                return 0;

        rc = foo_get_pre_ber(fe);
        if (rc < 0)
                return rc;
```

(continues on next page)

(continued from previous page)

```
        /* Check if post-BER stats are available */
        if (!(fe->status & FE_HAS_SYNC))
                return 0;

        rc = foo_get_post_ber(fe);
        if (rc < 0)
                return rc;
}

static const struct dvb_frontend_ops ops = {
        /* ... */
        .read_status = foo_get_status_and_stats,
};
```

Statistics collection

On almost all frontend hardware, the bit and byte counts are stored by the hardware after a certain amount of time or after the total bit/block counter reaches a certain value (usually programmable), for example, on every 1000 ms or after receiving 1,000,000 bits.

So, if you read the registers too soon, you'll end by reading the same value as in the previous reading, causing the monotonic value to be incremented too often.

Drivers should take the responsibility to avoid too often reads. That can be done using two approaches:

if the driver have a bit that indicates when a collected data is ready

Driver should check such bit before making the statistics available.

An example of such behavior can be found at this code snippet (adapted from mb86a20s driver's logic):

```
static int foo_get_pre_ber(struct dvb_frontend *fe)
{
        struct foo_state *state = fe->demodulator_priv;
        struct dtv_frontend_properties *c = &fe->dtv_property_cache;
        int rc, bit_error;

        /* Check if the BER measures are already available */
        rc = foo_read_u8(state, 0x54);
        if (rc < 0)
                return rc;

        if (!rc)
                return 0;

        /* Read Bit Error Count */
        bit_error = foo_read_u32(state, 0x55);
        if (bit_error < 0)
```

(continues on next page)

(continued from previous page)

```
                return bit_error;

        /* Read Total Bit Count */
        rc = foo_read_u32(state, 0x51);
        if (rc < 0)
                return rc;

        c->pre_bit_error.stat[0].scale = FE_SCALE_COUNTER;
        c->pre_bit_error.stat[0].uvalue += bit_error;
        c->pre_bit_count.stat[0].scale = FE_SCALE_COUNTER;
        c->pre_bit_count.stat[0].uvalue += rc;

        return 0;
}
```

If the driver doesn't provide a statistics available check bit

A few devices, however, may not provide a way to check if the stats are available (or the way to check it is unknown). They may not even provide a way to directly read the total number of bits or blocks.

On those devices, the driver need to ensure that it won't be reading from the register too often and/or estimate the total number of bits/blocks.

On such drivers, a typical routine to get statistics would be like (adapted from dib8000 driver's logic):

```
struct foo_state {
        /* ... */

        unsigned long per_jiffies_stats;
}

static int foo_get_pre_ber(struct dvb_frontend *fe)
{
        struct foo_state *state = fe->demodulator_priv;
        struct dtv_frontend_properties *c = &fe->dtv_property_cache;
        int rc, bit_error;
        u64 bits;

        /* Check if time for stats was elapsed */
        if (!time_after(jiffies, state->per_jiffies_stats))
                return 0;

        /* Next stat should be collected in 1000 ms */
        state->per_jiffies_stats = jiffies + msecs_to_jiffies(1000);

        /* Read Bit Error Count */
        bit_error = foo_read_u32(state, 0x55);
        if (bit_error < 0)
                return bit_error;

        /*
         * On this particular frontend, there's no register that
```

(continues on next page)

(continued from previous page)

```
         * would provide the number of bits per 1000ms sample. So,
         * some function would calculate it based on DTV properties
         */
        bits = get_number_of_bits_per_1000ms(fe);

        c->pre_bit_error.stat[0].scale = FE_SCALE_COUNTER;
        c->pre_bit_error.stat[0].uvalue += bit_error;
        c->pre_bit_count.stat[0].scale = FE_SCALE_COUNTER;
        c->pre_bit_count.stat[0].uvalue += bits;

        return 0;
}
```

Please notice that, on both cases, we' re getting the statistics using the `dvb_frontend_ops .read_status` callback. The rationale is that the frontend core will automatically call this function periodically (usually, 3 times per second, when the frontend is locked).

That warrants that we won't miss to collect a counter and increment the monotonic stats at the right time.

Digital TV Frontend functions and types

struct **dvb_frontend_tune_settings**
parameters to adjust frontend tuning

Definition

```
struct dvb_frontend_tune_settings {
  int min_delay_ms;
  int step_size;
  int max_drift;
};
```

Members

min_delay_ms minimum delay for tuning, in ms

step_size step size between two consecutive frequencies

max_drift maximum drift

NOTE

step_size is in Hz, for terrestrial/cable or kHz for satellite

struct **dvb_tuner_info**
Frontend name and min/max ranges/bandwidths

Definition

```
struct dvb_tuner_info {
  char name[128];
  u32 frequency_min_hz;
  u32 frequency_max_hz;
  u32 frequency_step_hz;
```

(continues on next page)

(continued from previous page)

```
  u32 bandwidth_min;
  u32 bandwidth_max;
  u32 bandwidth_step;
};
```

Members

name name of the Frontend

frequency_min_hz minimal frequency supported in Hz

frequency_max_hz maximum frequency supported in Hz

frequency_step_hz frequency step in Hz

bandwidth_min minimal frontend bandwidth supported

bandwidth_max maximum frontend bandwidth supported

bandwidth_step frontend bandwidth step

struct **analog_parameters**
Parameters to tune into an analog/radio channel

Definition

```
struct analog_parameters {
  unsigned int frequency;
  unsigned int mode;
  unsigned int audmode;
  u64 std;
};
```

Members

frequency Frequency used by analog TV tuner (either in 62.5 kHz step, for TV, or 62.5 Hz for radio)

mode Tuner mode, as defined on enum v4l2_tuner_type

audmode Audio mode as defined for the rxsubchans field at videodev2.h, e. g. V4L2_TUNER_MODE_*

std TV standard bitmap as defined at videodev2.h, e. g. V4L2_STD_*

Description

Hybrid tuners should be supported by both V4L2 and DVB APIs. This struct contains the data that are used by the V4L2 side. To avoid dependencies from V4L2 headers, all enums here are declared as integers.

enum **dvbfe_algo**
defines the algorithm used to tune into a channel

Constants

DVBFE_ALGO_HW Hardware Algorithm - Devices that support this algorithm do everything in hardware and no software support is needed to handle them. Requesting these devices to LOCK is the only thing required, device is supposed to do everything in the hardware.

DVBFE_ALGO_SW Software Algorithm - These are dumb devices, that require software to do everything

DVBFE_ALGO_CUSTOM Customizable Agorithm - Devices having this algorithm can be customized to have specific algorithms in the frontend driver, rather than simply doing a software zig-zag. In this case the zigzag maybe hardware assisted or it maybe completely done in hardware. In all cases, usage of this algorithm, in conjunction with the search and track callbacks, utilizes the driver specific algorithm.

DVBFE_ALGO_RECOVERY Recovery Algorithm - These devices have AUTO recovery capabilities from LOCK failure

enum **dvbfe_search**
search callback possible return status

Constants

DVBFE_ALGO_SEARCH_SUCCESS The frontend search algorithm completed and returned successfully

DVBFE_ALGO_SEARCH_ASLEEP The frontend search algorithm is sleeping

DVBFE_ALGO_SEARCH_FAILED The frontend search for a signal failed

DVBFE_ALGO_SEARCH_INVALID The frontend search algorithm was probably supplied with invalid parameters and the search is an invalid one

DVBFE_ALGO_SEARCH_AGAIN The frontend search algorithm was requested to search again

DVBFE_ALGO_SEARCH_ERROR The frontend search algorithm failed due to some error

struct **dvb_tuner_ops**
Tuner information and callbacks

Definition

```
struct dvb_tuner_ops {
  struct dvb_tuner_info info;
  void (*release)(struct dvb_frontend *fe);
  int (*init)(struct dvb_frontend *fe);
  int (*sleep)(struct dvb_frontend *fe);
  int (*suspend)(struct dvb_frontend *fe);
  int (*resume)(struct dvb_frontend *fe);
  int (*set_params)(struct dvb_frontend *fe);
  int (*set_analog_params)(struct dvb_frontend *fe, struct analog_
↪parameters *p);
  int (*set_config)(struct dvb_frontend *fe, void *priv_cfg);
  int (*get_frequency)(struct dvb_frontend *fe, u32 *frequency);
  int (*get_bandwidth)(struct dvb_frontend *fe, u32 *bandwidth);
  int (*get_if_frequency)(struct dvb_frontend *fe, u32 *frequency);
#define TUNER_STATUS_LOCKED 1;
#define TUNER_STATUS_STEREO 2;
  int (*get_status)(struct dvb_frontend *fe, u32 *status);
  int (*get_rf_strength)(struct dvb_frontend *fe, u16 *strength);
  int (*get_afc)(struct dvb_frontend *fe, s32 *afc);
  int (*calc_regs)(struct dvb_frontend *fe, u8 *buf, int buf_len);
```

(continues on next page)

(continued from previous page)

```
  int (*set_frequency)(struct dvb_frontend *fe, u32 frequency);
  int (*set_bandwidth)(struct dvb_frontend *fe, u32 bandwidth);
};
```

Members

info embedded `struct dvb_tuner_info` with tuner properties

release callback function called when frontend is detached. drivers should free any allocated memory.

init callback function used to initialize the tuner device.

sleep callback function used to put the tuner to sleep.

suspend callback function used to inform that the Kernel will suspend.

resume callback function used to inform that the Kernel is resuming from suspend.

set_params callback function used to inform the tuner to tune into a digital TV channel. The properties to be used are stored at `struct dvb_frontend`.dtv_property_cache. The tuner demod can change the parameters to reflect the changes needed for the channel to be tuned, and update statistics. This is the recommended way to set the tuner parameters and should be used on newer drivers.

set_analog_params callback function used to tune into an analog TV channel on hybrid tuners. It passes **analog_parameters** to the driver.

set_config callback function used to send some tuner-specific parameters.

get_frequency get the actual tuned frequency

get_bandwidth get the bandwidth used by the low pass filters

get_if_frequency get the Intermediate Frequency, in Hz. For baseband, should return 0.

get_status returns the frontend lock status

get_rf_strength returns the RF signal strength. Used mostly to support analog TV and radio. Digital TV should report, instead, via DVBv5 API (`struct dvb_frontend`.dtv_property_cache).

get_afc Used only by analog TV core. Reports the frequency drift due to AFC.

calc_regs callback function used to pass register data settings for simple tuners. Shouldn't be used on newer drivers.

set_frequency Set a new frequency. Shouldn't be used on newer drivers.

set_bandwidth Set a new frequency. Shouldn't be used on newer drivers.

NOTE

frequencies used on **get_frequency** and **set_frequency** are in Hz for terrestrial/cable or kHz for satellite.

struct **analog_demod_info**
Information struct for analog TV part of the demod

Definition

```
struct analog_demod_info {
  char *name;
};
```

Members

name Name of the analog TV demodulator

struct **analog_demod_ops**
Demodulation information and callbacks for analog TV and radio

Definition

```
struct analog_demod_ops {
  struct analog_demod_info info;
  void (*set_params)(struct dvb_frontend *fe, struct analog_parameters␣
↪*params);
  int (*has_signal)(struct dvb_frontend *fe, u16 *signal);
  int (*get_afc)(struct dvb_frontend *fe, s32 *afc);
  void (*tuner_status)(struct dvb_frontend *fe);
  void (*standby)(struct dvb_frontend *fe);
  void (*release)(struct dvb_frontend *fe);
  int (*i2c_gate_ctrl)(struct dvb_frontend *fe, int enable);
  int (*set_config)(struct dvb_frontend *fe, void *priv_cfg);
};
```

Members

info pointer to struct analog_demod_info

set_params callback function used to inform the demod to set the demodulator parameters needed to decode an analog or radio channel. The properties are passed via `struct analog_params`.

has_signal returns 0xffff if has signal, or 0 if it doesn't.

get_afc Used only by analog TV core. Reports the frequency drift due to AFC.

tuner_status callback function that returns tuner status bits, e. g. `TUNER_STATUS_LOCKED` and `TUNER_STATUS_STEREO`.

standby set the tuner to standby mode.

release callback function called when frontend is detached. drivers should free any allocated memory.

i2c_gate_ctrl controls the I2C gate. Newer drivers should use I2C mux support instead.

set_config callback function used to send some tuner-specific parameters.

struct **dvb_frontend_internal_info**
Frontend properties and capabilities

Definition

```
struct dvb_frontend_internal_info {
  char name[128];
```

(continues on next page)

(continued from previous page)

```
  u32 frequency_min_hz;
  u32 frequency_max_hz;
  u32 frequency_stepsize_hz;
  u32 frequency_tolerance_hz;
  u32 symbol_rate_min;
  u32 symbol_rate_max;
  u32 symbol_rate_tolerance;
  enum fe_caps caps;
};
```

Members

name Name of the frontend

frequency_min_hz Minimal frequency supported by the frontend.

frequency_max_hz Minimal frequency supported by the frontend.

frequency_stepsize_hz All frequencies are multiple of this value.

frequency_tolerance_hz Frequency tolerance.

symbol_rate_min Minimal symbol rate, in bauds (for Cable/Satellite systems).

symbol_rate_max Maximal symbol rate, in bauds (for Cable/Satellite systems).

symbol_rate_tolerance Maximal symbol rate tolerance, in ppm (for Cable/Satellite systems).

caps Capabilities supported by the frontend, as specified in `enum fe_caps`.

struct **dvb_frontend_ops**
Demodulation information and callbacks for ditialt TV

Definition

```
struct dvb_frontend_ops {
  struct dvb_frontend_internal_info info;
  u8 delsys[MAX_DELSYS];
  void (*detach)(struct dvb_frontend *fe);
  void (*release)(struct dvb_frontend* fe);
  void (*release_sec)(struct dvb_frontend* fe);
  int (*init)(struct dvb_frontend* fe);
  int (*sleep)(struct dvb_frontend* fe);
  int (*write)(struct dvb_frontend* fe, const u8 buf[], int len);
  int (*tune)(struct dvb_frontend* fe,bool re_tune,unsigned int mode_flags,
↪unsigned int *delay, enum fe_status *status);
  enum dvbfe_algo (*get_frontend_algo)(struct dvb_frontend *fe);
  int (*set_frontend)(struct dvb_frontend *fe);
  int (*get_tune_settings)(struct dvb_frontend* fe, struct dvb_frontend_
↪tune_settings* settings);
  int (*get_frontend)(struct dvb_frontend *fe, struct dtv_frontend_
↪properties *props);
  int (*read_status)(struct dvb_frontend *fe, enum fe_status *status);
  int (*read_ber)(struct dvb_frontend* fe, u32* ber);
  int (*read_signal_strength)(struct dvb_frontend* fe, u16* strength);
  int (*read_snr)(struct dvb_frontend* fe, u16* snr);
  int (*read_ucblocks)(struct dvb_frontend* fe, u32* ucblocks);
  int (*diseqc_reset_overload)(struct dvb_frontend* fe);
```

(continues on next page)

(continued from previous page)

```
  int (*diseqc_send_master_cmd)(struct dvb_frontend* fe, struct dvb_diseqc_
↪master_cmd* cmd);
  int (*diseqc_recv_slave_reply)(struct dvb_frontend* fe, struct dvb_
↪diseqc_slave_reply* reply);
  int (*diseqc_send_burst)(struct dvb_frontend *fe, enum fe_sec_mini_cmd␣
↪minicmd);
  int (*set_tone)(struct dvb_frontend *fe, enum fe_sec_tone_mode tone);
  int (*set_voltage)(struct dvb_frontend *fe, enum fe_sec_voltage voltage);
  int (*enable_high_lnb_voltage)(struct dvb_frontend* fe, long arg);
  int (*dishnetwork_send_legacy_command)(struct dvb_frontend* fe, unsigned␣
↪long cmd);
  int (*i2c_gate_ctrl)(struct dvb_frontend* fe, int enable);
  int (*ts_bus_ctrl)(struct dvb_frontend* fe, int acquire);
  int (*set_lna)(struct dvb_frontend *);
  enum dvbfe_search (*search)(struct dvb_frontend *fe);
  struct dvb_tuner_ops tuner_ops;
  struct analog_demod_ops analog_ops;
};
```

Members

info embedded `struct dvb_tuner_info` with tuner properties

delsys Delivery systems supported by the frontend

detach callback function called when frontend is detached. drivers should clean up, but not yet free the `struct dvb_frontend` allocation.

release callback function called when frontend is ready to be freed. drivers should free any allocated memory.

release_sec callback function requesting that the Satellite Equipment Control (SEC) driver to release and free any memory allocated by the driver.

init callback function used to initialize the tuner device.

sleep callback function used to put the tuner to sleep.

write callback function used by some demod legacy drivers to allow other drivers to write data into their registers. Should not be used on new drivers.

tune callback function used by demod drivers that use **DVBFE_ALGO_HW** to tune into a frequency.

get_frontend_algo returns the desired hardware algorithm.

set_frontend callback function used to inform the demod to set the parameters for demodulating a digital TV channel. The properties to be used are stored at `struct dvb_frontend`.dtv_property_cache. The demod can change the parameters to reflect the changes needed for the channel to be decoded, and update statistics.

get_tune_settings callback function

get_frontend callback function used to inform the parameters actuall in use. The properties to be used are stored at `struct dvb_frontend`.dtv_property_cache and update statistics. Please notice that it should not return an error code if the statistics are not available because the demog is not locked.

read_status returns the locking status of the frontend.

read_ber legacy callback function to return the bit error rate. Newer drivers should provide such info via DVBv5 API, e. g. **set_frontend**;/**get_frontend**, implementing this callback only if DVBv3 API compatibility is wanted.

read_signal_strength legacy callback function to return the signal strength. Newer drivers should provide such info via DVBv5 API, e. g. **set_frontend/get_frontend**, implementing this callback only if DVBv3 API compatibility is wanted.

read_snr legacy callback function to return the Signal/Noise rate. Newer drivers should provide such info via DVBv5 API, e. g. **set_frontend/get_frontend**, implementing this callback only if DVBv3 API compatibility is wanted.

read_ucblocks legacy callback function to return the Uncorrected Error Blocks. Newer drivers should provide such info via DVBv5 API, e. g. **set_frontend/get_frontend**, implementing this callback only if DVBv3 API compatibility is wanted.

diseqc_reset_overload callback function to implement the FE_DISEQC_RESET_OVERLOAD() ioctl (only Satellite)

diseqc_send_master_cmd callback function to implement the FE_DISEQC_SEND_MASTER_CMD() ioctl (only Satellite).

diseqc_recv_slave_reply callback function to implement the FE_DISEQC_RECV_SLAVE_REPLY() ioctl (only Satellite)

diseqc_send_burst callback function to implement the FE_DISEQC_SEND_BURST() ioctl (only Satellite).

set_tone callback function to implement the FE_SET_TONE() ioctl (only Satellite).

set_voltage callback function to implement the FE_SET_VOLTAGE() ioctl (only Satellite).

enable_high_lnb_voltage callback function to implement the FE_ENABLE_HIGH_LNB_VOLTAGE() ioctl (only Satellite).

dishnetwork_send_legacy_command callback function to implement the FE_DISHNETWORK_SEND_LEGACY_CMD() ioctl (only Satellite). Drivers should not use this, except when the DVB core emulation fails to provide proper support (e.g. if **set_voltage** takes more than 8ms to work), and when backward compatibility with this legacy API is required.

i2c_gate_ctrl controls the I2C gate. Newer drivers should use I2C mux support instead.

ts_bus_ctrl callback function used to take control of the TS bus.

set_lna callback function to power on/off/auto the LNA.

search callback function used on some custom algo search algos.

tuner_ops pointer to `struct dvb_tuner_ops`

analog_ops pointer to `struct analog_demod_ops`

struct **dtv_frontend_properties**
contains a list of properties that are specific to a digital TV standard.

Definition

```
struct dtv_frontend_properties {
  u32 frequency;
  enum fe_modulation      modulation;
  enum fe_sec_voltage     voltage;
  enum fe_sec_tone_mode   sectone;
  enum fe_spectral_inversion inversion;
  enum fe_code_rate       fec_inner;
  enum fe_transmit_mode   transmission_mode;
  u32 bandwidth_hz;
  enum fe_guard_interval  guard_interval;
  enum fe_hierarchy       hierarchy;
  u32 symbol_rate;
  enum fe_code_rate       code_rate_HP;
  enum fe_code_rate       code_rate_LP;
  enum fe_pilot           pilot;
  enum fe_rolloff         rolloff;
  enum fe_delivery_system delivery_system;
  enum fe_interleaving    interleaving;
  u8 isdbt_partial_reception;
  u8 isdbt_sb_mode;
  u8 isdbt_sb_subchannel;
  u32 isdbt_sb_segment_idx;
  u32 isdbt_sb_segment_count;
  u8 isdbt_layer_enabled;
  struct {
    u8 segment_count;
    enum fe_code_rate   fec;
    enum fe_modulation  modulation;
    u8 interleaving;
  } layer[3];
  u32 stream_id;
  u32 scrambling_sequence_index;
  u8 atscmh_fic_ver;
  u8 atscmh_parade_id;
  u8 atscmh_nog;
  u8 atscmh_tnog;
  u8 atscmh_sgn;
  u8 atscmh_prc;
  u8 atscmh_rs_frame_mode;
  u8 atscmh_rs_frame_ensemble;
  u8 atscmh_rs_code_mode_pri;
  u8 atscmh_rs_code_mode_sec;
  u8 atscmh_sccc_block_mode;
  u8 atscmh_sccc_code_mode_a;
  u8 atscmh_sccc_code_mode_b;
  u8 atscmh_sccc_code_mode_c;
  u8 atscmh_sccc_code_mode_d;
  u32 lna;
  struct dtv_fe_stats     strength;
  struct dtv_fe_stats     cnr;
  struct dtv_fe_stats     pre_bit_error;
  struct dtv_fe_stats     pre_bit_count;
```

(continues on next page)

(continued from previous page)

```
  struct dtv_fe_stats     post_bit_error;
  struct dtv_fe_stats     post_bit_count;
  struct dtv_fe_stats     block_error;
  struct dtv_fe_stats     block_count;
};
```

Members

frequency frequency in Hz for terrestrial/cable or in kHz for Satellite

modulation Frontend modulation type

voltage SEC voltage (only Satellite)

sectone SEC tone mode (only Satellite)

inversion Spectral inversion

fec_inner Forward error correction inner Code Rate

transmission_mode Transmission Mode

bandwidth_hz Bandwidth, in Hz. A zero value means that userspace wants to autodetect.

guard_interval Guard Interval

hierarchy Hierarchy

symbol_rate Symbol Rate

code_rate_HP high priority stream code rate

code_rate_LP low priority stream code rate

pilot Enable/disable/autodetect pilot tones

rolloff Rolloff factor (alpha)

delivery_system FE delivery system (e. g. digital TV standard)

interleaving interleaving

isdbt_partial_reception ISDB-T partial reception (only ISDB standard)

isdbt_sb_mode ISDB-T Sound Broadcast (SB) mode (only ISDB standard)

isdbt_sb_subchannel ISDB-T SB subchannel (only ISDB standard)

isdbt_sb_segment_idx ISDB-T SB segment index (only ISDB standard)

isdbt_sb_segment_count ISDB-T SB segment count (only ISDB standard)

isdbt_layer_enabled ISDB Layer enabled (only ISDB standard)

layer ISDB per-layer data (only ISDB standard)

layer.segment_count Segment Count;

layer.fec per layer code rate;

layer.modulation per layer modulation;

layer.interleaving per layer interleaving.

stream_id If different than zero, enable substream filtering, if hardware supports (DVB-S2 and DVB-T2).

scrambling_sequence_index Carries the index of the DVB-S2 physical layer scrambling sequence.

atscmh_fic_ver Version number of the FIC (Fast Information Channel) signaling data (only ATSC-M/H)

atscmh_parade_id Parade identification number (only ATSC-M/H)

atscmh_nog Number of MH groups per MH subframe for a designated parade (only ATSC-M/H)

atscmh_tnog Total number of MH groups including all MH groups belonging to all MH parades in one MH subframe (only ATSC-M/H)

atscmh_sgn Start group number (only ATSC-M/H)

atscmh_prc Parade repetition cycle (only ATSC-M/H)

atscmh_rs_frame_mode Reed Solomon (RS) frame mode (only ATSC-M/H)

atscmh_rs_frame_ensemble RS frame ensemble (only ATSC-M/H)

atscmh_rs_code_mode_pri RS code mode pri (only ATSC-M/H)

atscmh_rs_code_mode_sec RS code mode sec (only ATSC-M/H)

atscmh_sccc_block_mode Series Concatenated Convolutional Code (SCCC) Block Mode (only ATSC-M/H)

atscmh_sccc_code_mode_a SCCC code mode A (only ATSC-M/H)

atscmh_sccc_code_mode_b SCCC code mode B (only ATSC-M/H)

atscmh_sccc_code_mode_c SCCC code mode C (only ATSC-M/H)

atscmh_sccc_code_mode_d SCCC code mode D (only ATSC-M/H)

lna Power ON/OFF/AUTO the Linear Now-noise Amplifier (LNA)

strength DVBv5 API statistics: Signal Strength

cnr DVBv5 API statistics: Signal to Noise ratio of the (main) carrier

pre_bit_error DVBv5 API statistics: pre-Viterbi bit error count

pre_bit_count DVBv5 API statistics: pre-Viterbi bit count

post_bit_error DVBv5 API statistics: post-Viterbi bit error count

post_bit_count DVBv5 API statistics: post-Viterbi bit count

block_error DVBv5 API statistics: block error count

block_count DVBv5 API statistics: block count

NOTE

derivated statistics like Uncorrected Error blocks (UCE) are calculated on userspace.

Description

Only a subset of the properties are needed for a given delivery system. For more info, consult the media_api.html with the documentation of the Userspace API.

struct **dvb_frontend**
Frontend structure to be used on drivers.

Definition

```
struct dvb_frontend {
  struct kref refcount;
  struct dvb_frontend_ops ops;
  struct dvb_adapter *dvb;
  void *demodulator_priv;
  void *tuner_priv;
  void *frontend_priv;
  void *sec_priv;
  void *analog_demod_priv;
  struct dtv_frontend_properties dtv_property_cache;
#define DVB_FRONTEND_COMPONENT_TUNER 0;
#define DVB_FRONTEND_COMPONENT_DEMOD 1;
  int (*callback)(void *adapter_priv, int component, int cmd, int arg);
  int id;
  unsigned int exit;
};
```

Members

refcount refcount to keep track of `struct dvb_frontend` references

ops embedded `struct dvb_frontend_ops`

dvb pointer to `struct dvb_adapter`

demodulator_priv demod private data

tuner_priv tuner private data

frontend_priv frontend private data

sec_priv SEC private data

analog_demod_priv Analog demod private data

dtv_property_cache embedded `struct dtv_frontend_properties`

callback callback function used on some drivers to call either the tuner or the demodulator.

id Frontend ID

exit Used to inform the DVB core that the frontend thread should exit (usually, means that the hardware got disconnected.

int **dvb_register_frontend**(struct dvb_adapter * dvb, struct dvb_frontend * fe)
Registers a DVB frontend at the adapter

Parameters

struct dvb_adapter * dvb pointer to `struct dvb_adapter`

struct dvb_frontend * fe pointer to `struct dvb_frontend`

Description

Allocate and initialize the private data needed by the frontend core to manage the frontend and calls `dvb_register_device()` to register a new frontend. It also cleans the property cache that stores the frontend parameters and selects the first available delivery system.

int **dvb_unregister_frontend**(struct dvb_frontend * fe)
Unregisters a DVB frontend

Parameters

struct dvb_frontend * fe pointer to `struct dvb_frontend`

Description

Stops the frontend kthread, calls `dvb_unregister_device()` and frees the private frontend data allocated by `dvb_register_frontend()`.

NOTE

This function doesn' t frees the memory allocated by the demod, by the SEC driver and by the tuner. In order to free it, an explicit call to `dvb_frontend_detach()` is needed, after calling this function.

void **dvb_frontend_detach**(struct dvb_frontend * fe)
Detaches and frees frontend specific data

Parameters

struct dvb_frontend * fe pointer to `struct dvb_frontend`

Description

This function should be called after `dvb_unregister_frontend()`. It calls the SEC, tuner and demod release functions: `dvb_frontend_ops.release_sec`, `dvb_frontend_ops.tuner_ops`.release, `dvb_frontend_ops.analog_ops`.release and `dvb_frontend_ops.release`.

If the driver is compiled with `CONFIG_MEDIA_ATTACH`, it also decreases the module reference count, needed to allow userspace to remove the previously used DVB frontend modules.

int **dvb_frontend_suspend**(struct dvb_frontend * fe)
Suspends a Digital TV frontend

Parameters

struct dvb_frontend * fe pointer to `struct dvb_frontend`

Description

This function prepares a Digital TV frontend to suspend.

In order to prepare the tuner to suspend, if `dvb_frontend_ops.tuner_ops`.suspend() is available, it calls it. Otherwise, it will call `dvb_frontend_ops.tuner_ops`.sleep(), if available.

It will also call `dvb_frontend_ops.sleep`() to put the demod to suspend.

The drivers should also call `dvb_frontend_suspend()` as part of their handler for the `device_driver.suspend`().

int **dvb_frontend_resume**(struct dvb_frontend * fe)
Resumes a Digital TV frontend

Parameters

struct dvb_frontend * fe pointer to `struct dvb_frontend`

Description

This function resumes the usual operation of the tuner after resume.

In order to resume the frontend, it calls the demod `dvb_frontend_ops.init`().

If `dvb_frontend_ops.tuner_ops`.resume() is available, It, it calls it. Otherwise,t will call `dvb_frontend_ops.tuner_ops`.init(), if available.

Once tuner and demods are resumed, it will enforce that the SEC voltage and tone are restored to their previous values and wake up the frontend's kthread in order to retune the frontend.

The drivers should also call `dvb_frontend_resume()` as part of their handler for the `device_driver.resume`().

void **dvb_frontend_reinitialise**(struct dvb_frontend * fe)
forces a reinitialisation at the frontend

Parameters

struct dvb_frontend * fe pointer to `struct dvb_frontend`

Description

Calls `dvb_frontend_ops.init`() and `dvb_frontend_ops.tuner_ops`.init(), and resets SEC tone and voltage (for Satellite systems).

NOTE

Currently, this function is used only by one driver (budget-av). It seems to be due to address some special issue with that specific frontend.

void **dvb_frontend_sleep_until**(ktime_t * waketime, u32 add_usec)
Sleep for the amount of time given by add_usec parameter

Parameters

ktime_t * waketime pointer to `struct ktime_t`

u32 add_usec time to sleep, in microseconds

Description

This function is used to measure the time required for the FE_DISHNETWORK_SEND_LEGACY_CMD() ioctl to work. It needs to be as precise as possible, as it affects the detection of the dish tone command at the satellite subsystem.

Its used internally by the DVB frontend core, in order to emulate FE_DISHNETWORK_SEND_LEGACY_CMD() using the `dvb_frontend_ops.set_voltage`() callback.

NOTE

it should not be used at the drivers, as the emulation for the legacy callback is provided by the Kernel. The only situation where this should be at the drivers is when there are some bugs at the hardware that would prevent the core emulation to work. On such cases, the driver would be writing a `dvb_frontend_ops.dishnetwork_send_legacy_command`() and calling this function directly.

53.2.3 Digital TV Demux kABI

Digital TV Demux

The Kernel Digital TV Demux kABI defines a driver-internal interface for registering low-level, hardware specific driver to a hardware independent demux layer. It is only of interest for Digital TV device driver writers. The header file for this kABI is named `demux.h` and located in `include/media`.

The demux kABI should be implemented for each demux in the system. It is used to select the TS source of a demux and to manage the demux resources. When the demux client allocates a resource via the demux kABI, it receives a pointer to the kABI of that resource.

Each demux receives its TS input from a DVB front-end or from memory, as set via this demux kABI. In a system with more than one front-end, the kABI can be used to select one of the DVB front-ends as a TS source for a demux, unless this is fixed in the HW platform.

The demux kABI only controls front-ends regarding to their connections with demuxes; the kABI used to set the other front-end parameters, such as tuning, are devined via the Digital TV Frontend kABI.

The functions that implement the abstract interface demux should be defined static or module private and registered to the Demux core for external access. It is not necessary to implement every function in the struct `dmx_demux`. For example, a demux interface might support Section filtering, but not PES filtering. The kABI client is expected to check the value of any function pointer before calling the function: the value of `NULL` means that the function is not available.

Whenever the functions of the demux API modify shared data, the possibilities of lost update and race condition problems should be addressed, e.g. by protecting parts of code with mutexes.

Note that functions called from a bottom half context must not sleep. Even a simple memory allocation without using `GFP_ATOMIC` can result in a kernel thread being put to sleep if swapping is needed. For example, the Linux Kernel calls the functions of a network device interface from a bottom half context. Thus, if a demux kABI function is called from network device code, the function must not sleep.

Demux Callback API

This kernel-space API comprises the callback functions that deliver filtered data to the demux client. Unlike the other DVB kABIs, these functions are provided by the client and called from the demux code.

The function pointers of this abstract interface are not packed into a structure as in the other demux APIs, because the callback functions are registered and used independent of each other. As an example, it is possible for the API client to provide several callback functions for receiving TS packets and no callbacks for PES packets or sections.

The functions that implement the callback API need not be re-entrant: when a demux driver calls one of these functions, the driver is not allowed to call the function again before the original call returns. If a callback is triggered by a hardware interrupt, it is recommended to use the Linux bottom half mechanism or start a tasklet instead of making the callback function call directly from a hardware interrupt.

This mechanism is implemented by `dmx_ts_cb()` and `dmx_section_cb()` callbacks.

Digital TV Demux device registration functions and data structures

enum **dmxdev_type**
type of demux filter type.

Constants

DMXDEV_TYPE_NONE no filter set.

DMXDEV_TYPE_SEC section filter.

DMXDEV_TYPE_PES Program Elementary Stream (PES) filter.

enum **dmxdev_state**
state machine for the dmxdev.

Constants

DMXDEV_STATE_FREE indicates that the filter is freed.

DMXDEV_STATE_ALLOCATED indicates that the filter was allocated to be used.

DMXDEV_STATE_SET indicates that the filter parameters are set.

DMXDEV_STATE_GO indicates that the filter is running.

DMXDEV_STATE_DONE indicates that a packet was already filtered and the filter is now disabled. Set only if `DMX_ONESHOT`. See `dmx_sct_filter_params`.

DMXDEV_STATE_TIMEDOUT Indicates a timeout condition.

struct **dmxdev_feed**
digital TV dmxdev feed

Definition

```
struct dmxdev_feed {
  u16 pid;
  struct dmx_ts_feed *ts;
  struct list_head next;
};
```

Members

pid Program ID to be filtered

ts pointer to `struct dmx_ts_feed`

next `struct list_head` pointing to the next feed.

struct **dmxdev_filter**
digital TV dmxdev filter

Definition

```
struct dmxdev_filter {
  union {
    struct dmx_section_filter *sec;
  } filter;
  union {
    struct list_head ts;
    struct dmx_section_feed *sec;
  } feed;
  union {
    struct dmx_sct_filter_params sec;
    struct dmx_pes_filter_params pes;
  } params;
  enum dmxdev_type type;
  enum dmxdev_state state;
  struct dmxdev *dev;
  struct dvb_ringbuffer buffer;
  struct dvb_vb2_ctx vb2_ctx;
  struct mutex mutex;
  struct timer_list timer;
  int todo;
  u8 secheader[3];
};
```

Members

filter a union describing a dmxdev filter. Currently used only for section filters.

filter.sec a `struct dmx_section_filter` pointer. For section filter only.

feed a union describing a dmxdev feed. Depending on the filter type, it can be either **feed.ts** or **feed.sec**.

feed.ts a `struct list_head` list. For TS and PES feeds.

feed.sec a `struct dmx_section_feed` pointer. For section feed only.

params a union describing dmxdev filter parameters. Depending on the filter type, it can be either **params.sec** or **params.pes**.

params.sec a `struct dmx_sct_filter_params` embedded struct. For section filter only.

params.pes a `struct dmx_pes_filter_params` embedded struct. For PES filter only.

type type of the dmxdev filter, as defined by `enum dmxdev_type`.

state state of the dmxdev filter, as defined by `enum dmxdev_state`.

dev pointer to `struct dmxdev`.

buffer an embedded `struct dvb_ringbuffer` buffer.

vb2_ctx control struct for VB2 handler

mutex protects the access to `struct dmxdev_filter`.

timer `struct timer_list` embedded timer, used to check for feed timeouts. Only for section filter.

todo index for the **secheader**. Only for section filter.

secheader buffer cache to parse the section header. Only for section filter.

struct **dmxdev**
Describes a digital TV demux device.

Definition

```
struct dmxdev {
  struct dvb_device *dvbdev;
  struct dvb_device *dvr_dvbdev;
  struct dmxdev_filter *filter;
  struct dmx_demux *demux;
  int filternum;
  int capabilities;
  unsigned int may_do_mmap:1;
  unsigned int exit:1;
#define DMXDEV_CAP_DUPLEX 1;
  struct dmx_frontend *dvr_orig_fe;
  struct dvb_ringbuffer dvr_buffer;
#define DVR_BUFFER_SIZE (10*188*1024);
  struct dvb_vb2_ctx dvr_vb2_ctx;
  struct mutex mutex;
  spinlock_t lock;
};
```

Members

dvbdev pointer to `struct dvb_device` associated with the demux device node.

dvr_dvbdev pointer to `struct dvb_device` associated with the dvr device node.

filter pointer to `struct dmxdev_filter`.

demux pointer to `struct dmx_demux`.

filternum number of filters.

capabilities demux capabilities as defined by `enum dmx_demux_caps`.

may_do_mmap flag used to indicate if the device may do mmap.

exit flag to indicate that the demux is being released.

dvr_orig_fe pointer to `struct dmx_frontend`.

dvr_buffer embedded `struct dvb_ringbuffer` for DVB output.

dvr_vb2_ctx control struct for VB2 handler

mutex protects the usage of this structure.

lock protects access to `dmxdev->filter->`data.

int **dvb_dmxdev_init**(struct dmxdev * dmxdev, struct dvb_adapter * adap)
initializes a digital TV demux and registers both demux and DVR devices.

Parameters

struct dmxdev * dmxdev pointer to `struct dmxdev`.

struct dvb_adapter * adap pointer to `struct dvb_adapter`.

void **dvb_dmxdev_release**(struct dmxdev * dmxdev)
releases a digital TV demux and unregisters it.

Parameters

struct dmxdev * dmxdev pointer to `struct dmxdev`.

High-level Digital TV demux interface

enum **dvb_dmx_filter_type**
type of demux feed.

Constants

DMX_TYPE_TS feed is in TS mode.

DMX_TYPE_SEC feed is in Section mode.

enum **dvb_dmx_state**
state machine for a demux filter.

Constants

DMX_STATE_FREE indicates that the filter is freed.

DMX_STATE_ALLOCATED indicates that the filter was allocated to be used.

DMX_STATE_READY indicates that the filter is ready to be used.

DMX_STATE_GO indicates that the filter is running.

struct **dvb_demux_filter**
Describes a DVB demux section filter.

Definition

```
struct dvb_demux_filter {
  struct dmx_section_filter filter;
  u8 maskandmode[DMX_MAX_FILTER_SIZE];
  u8 maskandnotmode[DMX_MAX_FILTER_SIZE];
  bool doneq;
  struct dvb_demux_filter *next;
  struct dvb_demux_feed *feed;
```

(continues on next page)

(continued from previous page)

```
  int index;
  enum dvb_dmx_state state;
  enum dvb_dmx_filter_type type;
};
```

Members

filter Section filter as defined by `struct dmx_section_filter`.

maskandmode logical `and` bit mask.

maskandnotmode logical `and not` bit mask.

doneq flag that indicates when a filter is ready.

next pointer to the next section filter.

feed `struct dvb_demux_feed` pointer.

index index of the used demux filter.

state state of the filter as described by `enum dvb_dmx_state`.

type type of the filter as described by `enum dvb_dmx_filter_type`.

struct **dvb_demux_feed**
describes a DVB field

Definition

```
struct dvb_demux_feed {
  union {
    struct dmx_ts_feed ts;
    struct dmx_section_feed sec;
  } feed;
  union {
    dmx_ts_cb ts;
    dmx_section_cb sec;
  } cb;
  struct dvb_demux *demux;
  void *priv;
  enum dvb_dmx_filter_type type;
  enum dvb_dmx_state state;
  u16 pid;
  ktime_t timeout;
  struct dvb_demux_filter *filter;
  u32 buffer_flags;
  enum ts_filter_type ts_type;
  enum dmx_ts_pes pes_type;
  int cc;
  bool pusi_seen;
  u16 peslen;
  struct list_head list_head;
  unsigned int index;
};
```

Members

feed a union describing a digital TV feed. Depending on the feed type, it can be either **feed.ts** or **feed.sec**.

feed.ts a `struct dmx_ts_feed` pointer. For TS feed only.

feed.sec a `struct dmx_section_feed` pointer. For section feed only.

cb a union describing digital TV callbacks. Depending on the feed type, it can be either **cb.ts** or **cb.sec**.

cb.ts a `dmx_ts_cb()` calback function pointer. For TS feed only.

cb.sec a `dmx_section_cb()` callback function pointer. For section feed only.

demux pointer to `struct dvb_demux`.

priv private data that can optionally be used by a DVB driver.

type type of the filter, as defined by `enum dvb_dmx_filter_type`.

state state of the filter as defined by `enum dvb_dmx_state`.

pid PID to be filtered.

timeout feed timeout.

filter pointer to `struct dvb_demux_filter`.

buffer_flags Buffer flags used to report discontinuity users via DVB memory mapped API, as defined by `enum dmx_buffer_flags`.

ts_type type of TS, as defined by `enum ts_filter_type`.

pes_type type of PES, as defined by `enum dmx_ts_pes`.

cc MPEG-TS packet continuity counter

pusi_seen if true, indicates that a discontinuity was detected. it is used to prevent feeding of garbage from previous section.

peslen length of the PES (Packet Elementary Stream).

list_head head for the list of digital TV demux feeds.

index a unique index for each feed. Can be used as hardware pid filter index.

struct **dvb_demux**
represents a digital TV demux

Definition

```
struct dvb_demux {
  struct dmx_demux dmx;
  void *priv;
  int filternum;
  int feednum;
  int (*start_feed)(struct dvb_demux_feed *feed);
  int (*stop_feed)(struct dvb_demux_feed *feed);
  int (*write_to_decoder)(struct dvb_demux_feed *feed, const u8 *buf, size_
↪t len);
  u32 (*check_crc32)(struct dvb_demux_feed *feed, const u8 *buf, size_t␣
↪len);
  void (*memcopy)(struct dvb_demux_feed *feed, u8 *dst, const u8 *src,␣
↪size_t len);
  int users;
#define MAX_DVB_DEMUX_USERS 10;
```

(continues on next page)

(continued from previous page)

```
  struct dvb_demux_filter *filter;
  struct dvb_demux_feed *feed;
  struct list_head frontend_list;
  struct dvb_demux_feed *pesfilter[DMX_PES_OTHER];
  u16 pids[DMX_PES_OTHER];
#define DMX_MAX_PID 0x2000;
  struct list_head feed_list;
  u8 tsbuf[204];
  int tsbufp;
  struct mutex mutex;
  spinlock_t lock;
  uint8_t *cnt_storage;
  ktime_t speed_last_time;
  uint32_t speed_pkts_cnt;
};
```

Members

dmx embedded `struct dmx_demux` with demux capabilities and callbacks.

priv private data that can optionally be used by a DVB driver.

filternum maximum amount of DVB filters.

feednum maximum amount of DVB feeds.

start_feed callback routine to be called in order to start a DVB feed.

stop_feed callback routine to be called in order to stop a DVB feed.

write_to_decoder callback routine to be called if the feed is TS and it is routed to an A/V decoder, when a new TS packet is received. Used only on av7110-av.c.

check_crc32 callback routine to check CRC. If not initialized, dvb_demux will use an internal one.

memcopy callback routine to memcopy received data. If not initialized, dvb_demux will default to memcpy().

users counter for the number of demux opened file descriptors. Currently, it is limited to 10 users.

filter pointer to `struct dvb_demux_filter`.

feed pointer to `struct dvb_demux_feed`.

frontend_list `struct list_head` with frontends used by the demux.

pesfilter array of `struct dvb_demux_feed` with the PES types that will be filtered.

pids list of filtered program IDs.

feed_list `struct list_head` with feeds.

tsbuf temporary buffer used internally to store TS packets.

tsbufp temporary buffer index used internally.

mutex pointer to `struct mutex` used to protect feed set logic.

lock pointer to `spinlock_t`, used to protect buffer handling.

cnt_storage buffer used for TS/TEI continuity check.

speed_last_time ktime_t used for TS speed check.

speed_pkts_cnt packets count used for TS speed check.

int **dvb_dmx_init**(struct dvb_demux * demux)
initialize a digital TV demux struct.

Parameters

struct dvb_demux * demux struct dvb_demux to be initialized.

Description

Before being able to register a digital TV demux struct, drivers should call this routine. On its typical usage, some fields should be initialized at the driver before calling it.

A typical usecase is:

```
dvb->demux.dmx.capabilities =
        DMX_TS_FILTERING | DMX_SECTION_FILTERING |
        DMX_MEMORY_BASED_FILTERING;
dvb->demux.priv       = dvb;
dvb->demux.filternum  = 256;
dvb->demux.feednum    = 256;
dvb->demux.start_feed = driver_start_feed;
dvb->demux.stop_feed  = driver_stop_feed;
ret = dvb_dmx_init(&dvb->demux);
if (ret < 0)
        return ret;
```

void **dvb_dmx_release**(struct dvb_demux * demux)
releases a digital TV demux internal buffers.

Parameters

struct dvb_demux * demux struct dvb_demux to be released.

Description

The DVB core internally allocates data at **demux**. This routine releases those data. Please notice that the struct itelf is not released, as it can be embedded on other structs.

void **dvb_dmx_swfilter_packets**(struct dvb_demux * demux, const u8 * buf,
size_t count)
use dvb software filter for a buffer with multiple MPEG-TS packets with 188 bytes each.

Parameters

struct dvb_demux * demux pointer to struct dvb_demux

const u8 * buf buffer with data to be filtered

size_t count number of MPEG-TS packets with size of 188.

Description

The routine will discard a DVB packet that don’t start with 0x47.

Use this routine if the DVB demux fills MPEG-TS buffers that are already aligned.

NOTE

The **buf** size should have size equal to `count * 188`.

void **dvb_dmx_swfilter**(struct dvb_demux * demux, const u8 * buf, size_t count)

use dvb software filter for a buffer with multiple MPEG-TS packets with 188 bytes each.

Parameters

`struct dvb_demux * demux` pointer to `struct dvb_demux`

`const u8 * buf` buffer with data to be filtered

`size_t count` number of MPEG-TS packets with size of 188.

Description

If a DVB packet doesn't start with 0x47, it will seek for the first byte that starts with 0x47.

Use this routine if the DVB demux fill buffers that may not start with a packet start mark (0x47).

NOTE

The **buf** size should have size equal to `count * 188`.

void **dvb_dmx_swfilter_204**(struct dvb_demux * demux, const u8 * buf, size_t count)

use dvb software filter for a buffer with multiple MPEG-TS packets with 204 bytes each.

Parameters

`struct dvb_demux * demux` pointer to `struct dvb_demux`

`const u8 * buf` buffer with data to be filtered

`size_t count` number of MPEG-TS packets with size of 204.

Description

If a DVB packet doesn't start with 0x47, it will seek for the first byte that starts with 0x47.

Use this routine if the DVB demux fill buffers that may not start with a packet start mark (0x47).

NOTE

The **buf** size should have size equal to `count * 204`.

void **dvb_dmx_swfilter_raw**(struct dvb_demux * demux, const u8 * buf, size_t count)

make the raw data available to userspace without filtering

Parameters

`struct dvb_demux * demux` pointer to `struct dvb_demux`

`const u8 * buf` buffer with data

size_t count number of packets to be passed. The actual size of each packet depends on the dvb_demux->feed->cb.ts logic.

Description

Use it if the driver needs to deliver the raw payload to userspace without passing through the kernel demux. That is meant to support some delivery systems that aren't based on MPEG-TS.

This function relies on dvb_demux->feed->cb.ts to actually handle the buffer.

Driver-internal low-level hardware specific driver demux interface

enum **ts_filter_type**
filter type bitmap for dmx_ts_feed.set()

Constants

TS_PACKET Send TS packets (188 bytes) to callback (default).

TS_PAYLOAD_ONLY In case TS_PACKET is set, only send the TS payload (<=184 bytes per packet) to callback

TS_DECODER Send stream to built-in decoder (if present).

TS_DEMUX In case TS_PACKET is set, send the TS to the demux device, not to the dvr device

struct **dmx_ts_feed**
Structure that contains a TS feed filter

Definition

```
struct dmx_ts_feed {
  int is_filtering;
  struct dmx_demux *parent;
  void *priv;
  int (*set)(struct dmx_ts_feed *feed,u16 pid,int type,enum dmx_ts_pes pes_
↪type, ktime_t timeout);
  int (*start_filtering)(struct dmx_ts_feed *feed);
  int (*stop_filtering)(struct dmx_ts_feed *feed);
};
```

Members

is_filtering Set to non-zero when filtering in progress

parent pointer to struct dmx_demux

priv pointer to private data of the API client

set sets the TS filter

start_filtering starts TS filtering

stop_filtering stops TS filtering

Description

A TS feed is typically mapped to a hardware PID filter on the demux chip. Using this API, the client can set the filtering properties to start/stop filtering TS packets on a particular TS feed.

struct **dmx_section_filter**
Structure that describes a section filter

Definition

```
struct dmx_section_filter {
  u8 filter_value[DMX_MAX_FILTER_SIZE];
  u8 filter_mask[DMX_MAX_FILTER_SIZE];
  u8 filter_mode[DMX_MAX_FILTER_SIZE];
  struct dmx_section_feed *parent;
  void *priv;
};
```

Members

filter_value Contains up to 16 bytes (128 bits) of the TS section header that will be matched by the section filter

filter_mask Contains a 16 bytes (128 bits) filter mask with the bits specified by **filter_value** that will be used on the filter match logic.

filter_mode Contains a 16 bytes (128 bits) filter mode.

parent Back-pointer to struct dmx_section_feed.

priv Pointer to private data of the API client.

Description

The **filter_mask** controls which bits of **filter_value** are compared with the section headers/payload. On a binary value of 1 in filter_mask, the corresponding bits are compared. The filter only accepts sections that are equal to filter_value in all the tested bit positions.

struct **dmx_section_feed**
Structure that contains a section feed filter

Definition

```
struct dmx_section_feed {
  int is_filtering;
  struct dmx_demux *parent;
  void *priv;
  int check_crc;
  int (*set)(struct dmx_section_feed *feed,u16 pid, int check_crc);
  int (*allocate_filter)(struct dmx_section_feed *feed, struct dmx_section_
↪filter **filter);
  int (*release_filter)(struct dmx_section_feed *feed, struct dmx_section_
↪filter *filter);
  int (*start_filtering)(struct dmx_section_feed *feed);
  int (*stop_filtering)(struct dmx_section_feed *feed);
};
```

Members

is_filtering Set to non-zero when filtering in progress

parent pointer to struct dmx_demux

priv pointer to private data of the API client

check_crc If non-zero, check the CRC values of filtered sections.

set sets the section filter

allocate_filter This function is used to allocate a section filter on the demux. It should only be called when no filtering is in progress on this section feed. If a filter cannot be allocated, the function fails with -ENOSPC.

release_filter This function releases all the resources of a previously allocated section filter. The function should not be called while filtering is in progress on this section feed. After calling this function, the caller should not try to dereference the filter pointer.

start_filtering starts section filtering

stop_filtering stops section filtering

Description

A TS feed is typically mapped to a hardware PID filter on the demux chip. Using this API, the client can set the filtering properties to start/stop filtering TS packets on a particular TS feed.

dmx_ts_cb
Typedef: DVB demux TS filter callback function prototype

Syntax

```
int dmx_ts_cb (const u8 * buffer1, size_t buffer1_length,
const u8 * buffer2, size_t buffer2_length, struct
dmx_ts_feed * source, u32 * buffer_flags);
```

Parameters

const u8 * buffer1 Pointer to the start of the filtered TS packets.

size_t buffer1_length Length of the TS data in buffer1.

const u8 * buffer2 Pointer to the tail of the filtered TS packets, or NULL.

size_t buffer2_length Length of the TS data in buffer2.

struct dmx_ts_feed * source Indicates which TS feed is the source of the callback.

u32 * buffer_flags Address where buffer flags are stored. Those are used to report discontinuity users via DVB memory mapped API, as defined by enum dmx_buffer_flags.

Description

This function callback prototype, provided by the client of the demux API, is called from the demux code. The function is only called when filtering on a TS feed has been enabled using the start_filtering() function at the dmx_demux. Any TS packets that match the filter settings are copied to a circular buffer. The filtered TS packets are delivered to the client using this callback function. It is expected that the **buffer1** and **buffer2** callback parameters point to addresses within the circular buffer, but other implementations are also possible. Note that the called

party should not try to free the memory the **buffer1** and **buffer2** parameters point to.

When this function is called, the **buffer1** parameter typically points to the start of the first undelivered TS packet within a circular buffer. The **buffer2** buffer parameter is normally NULL, except when the received TS packets have crossed the last address of the circular buffer and "wrapped" to the beginning of the buffer. In the latter case the **buffer1** parameter would contain an address within the circular buffer, while the **buffer2** parameter would contain the first address of the circular buffer. The number of bytes delivered with this function (i.e. **buffer1_length** + **buffer2_length**) is usually equal to the value of callback_length parameter given in the set() function, with one exception: if a timeout occurs before receiving callback_length bytes of TS data, any undelivered packets are immediately delivered to the client by calling this function. The timeout duration is controlled by the set() function in the TS Feed API.

If a TS packet is received with errors that could not be fixed by the TS-level forward error correction (FEC), the Transport_error_indicator flag of the TS packet header should be set. The TS packet should not be discarded, as the error can possibly be corrected by a higher layer protocol. If the called party is slow in processing the callback, it is possible that the circular buffer eventually fills up. If this happens, the demux driver should discard any TS packets received while the buffer is full and return -EOVERFLOW.

The type of data returned to the callback can be selected by the `dmx_ts_feed`.**set** function. The type parameter decides if the raw TS packet (TS_PACKET) or just the payload (TS_PACKET|TS_PAYLOAD_ONLY) should be returned. If additionally the TS_DECODER bit is set the stream will also be sent to the hardware MPEG decoder.

- 0, on success;
- -EOVERFLOW, on buffer overflow.

Return

`dmx_section_cb`
Typedef: DVB demux TS filter callback function prototype

Syntax

```
int dmx_section_cb (const u8 * buffer1, size_t
buffer1_len, const u8 * buffer2, size_t buffer2_len, struct
dmx_section_filter * source, u32 * buffer_flags);
```

Parameters

`const u8 * buffer1` Pointer to the start of the filtered section, e.g. within the circular buffer of the demux driver.

`size_t buffer1_len` Length of the filtered section data in **buffer1**, including headers and CRC.

`const u8 * buffer2` Pointer to the tail of the filtered section data, or NULL. Useful to handle the wrapping of a circular buffer.

`size_t buffer2_len` Length of the filtered section data in **buffer2**, including headers and CRC.

struct dmx_section_filter * source Indicates which section feed is the source of the callback.

u32 * buffer_flags Address where buffer flags are stored. Those are used to report discontinuity users via DVB memory mapped API, as defined by `enum dmx_buffer_flags`.

Description

This function callback prototype, provided by the client of the demux API, is called from the demux code. The function is only called when filtering of sections has been enabled using the function `dmx_ts_feed`.**start_filtering**. When the demux driver has received a complete section that matches at least one section filter, the client is notified via this callback function. Normally this function is called for each received section; however, it is also possible to deliver multiple sections with one callback, for example when the system load is high. If an error occurs while receiving a section, this function should be called with the corresponding error type set in the success field, whether or not there is data to deliver. The Section Feed implementation should maintain a circular buffer for received sections. However, this is not necessary if the Section Feed API is implemented as a client of the TS Feed API, because the TS Feed implementation then buffers the received data. The size of the circular buffer can be configured using the `dmx_ts_feed`.**set** function in the Section Feed API. If there is no room in the circular buffer when a new section is received, the section must be discarded. If this happens, the value of the success parameter should be DMX_OVERRUN_ERROR on the next callback.

enum **dmx_frontend_source**
Used to identify the type of frontend

Constants

DMX_MEMORY_FE The source of the demux is memory. It means that the MPEG-TS to be filtered comes from userspace, via write() syscall.

DMX_FRONTEND_0 The source of the demux is a frontend connected to the demux.

struct **dmx_frontend**
Structure that lists the frontends associated with a demux

Definition

```
struct dmx_frontend {
  struct list_head connectivity_list;
  enum dmx_frontend_source source;
};
```

Members

connectivity_list List of front-ends that can be connected to a particular demux;

source Type of the frontend.

Description

FIXME: this structure should likely be replaced soon by some media-controller based logic.

enum **dmx_demux_caps**
MPEG-2 TS Demux capabilities bitmap

Constants

DMX_TS_FILTERING set if TS filtering is supported;

DMX_SECTION_FILTERING set if section filtering is supported;

DMX_MEMORY_BASED_FILTERING set if write() available.

Description

Those flags are OR'ed in the `dmx_demux.capabilities` field

DMX_FE_ENTRY(list)
Casts elements in the list of registered front-ends from the generic type struct list_head to the type * struct dmx_frontend

Parameters

list list of struct dmx_frontend

struct **dmx_demux**
Structure that contains the demux capabilities and callbacks.

Definition

```
struct dmx_demux {
  enum dmx_demux_caps capabilities;
  struct dmx_frontend *frontend;
  void *priv;
  int (*open)(struct dmx_demux *demux);
  int (*close)(struct dmx_demux *demux);
  int (*write)(struct dmx_demux *demux, const char __user *buf, size_t␣
↪count);
  int (*allocate_ts_feed)(struct dmx_demux *demux,struct dmx_ts_feed␣
↪**feed, dmx_ts_cb callback);
  int (*release_ts_feed)(struct dmx_demux *demux, struct dmx_ts_feed␣
↪*feed);
  int (*allocate_section_feed)(struct dmx_demux *demux,struct dmx_section_
↪feed **feed, dmx_section_cb callback);
  int (*release_section_feed)(struct dmx_demux *demux, struct dmx_section_
↪feed *feed);
  int (*add_frontend)(struct dmx_demux *demux, struct dmx_frontend␣
↪*frontend);
  int (*remove_frontend)(struct dmx_demux *demux, struct dmx_frontend␣
↪*frontend);
  struct list_head *(*get_frontends)(struct dmx_demux *demux);
  int (*connect_frontend)(struct dmx_demux *demux, struct dmx_frontend␣
↪*frontend);
  int (*disconnect_frontend)(struct dmx_demux *demux);
  int (*get_pes_pids)(struct dmx_demux *demux, u16 *pids);
};
```

Members

capabilities Bitfield of capability flags.

frontend Front-end connected to the demux

priv Pointer to private data of the API client

open This function reserves the demux for use by the caller and, if necessary, initializes the demux. When the demux is no longer needed, the function **close** should be called. It should be possible for multiple clients to access the demux at the same time. Thus, the function implementation should increment the demux usage count when **open** is called and decrement it when **close** is called. The **demux** function parameter contains a pointer to the demux API and instance data. It returns: 0 on success; -EUSERS, if maximum usage count was reached; -EINVAL, on bad parameter.

close This function reserves the demux for use by the caller and, if necessary, initializes the demux. When the demux is no longer needed, the function **close** should be called. It should be possible for multiple clients to access the demux at the same time. Thus, the function implementation should increment the demux usage count when **open** is called and decrement it when **close** is called. The **demux** function parameter contains a pointer to the demux API and instance data. It returns: 0 on success; -ENODEV, if demux was not in use (e. g. no users); -EINVAL, on bad parameter.

write This function provides the demux driver with a memory buffer containing TS packets. Instead of receiving TS packets from the DVB front-end, the demux driver software will read packets from memory. Any clients of this demux with active TS, PES or Section filters will receive filtered data via the Demux callback API (see 0). The function returns when all the data in the buffer has been consumed by the demux. Demux hardware typically cannot read TS from memory. If this is the case, memory-based filtering has to be implemented entirely in software. The **demux** function parameter contains a pointer to the demux API and instance data. The **buf** function parameter contains a pointer to the TS data in kernel-space memory. The **count** function parameter contains the length of the TS data. It returns: 0 on success; -ERESTARTSYS, if mutex lock was interrupted; -EINTR, if a signal handling is pending; -ENODEV, if demux was removed; -EINVAL, on bad parameter.

allocate_ts_feed Allocates a new TS feed, which is used to filter the TS packets carrying a certain PID. The TS feed normally corresponds to a hardware PID filter on the demux chip. The **demux** function parameter contains a pointer to the demux API and instance data. The **feed** function parameter contains a pointer to the TS feed API and instance data. The **callback** function parameter contains a pointer to the callback function for passing received TS packet. It returns: 0 on success; -ERESTARTSYS, if mutex lock was interrupted; -EBUSY, if no more TS feeds is available; -EINVAL, on bad parameter.

release_ts_feed Releases the resources allocated with **allocate_ts_feed**. Any filtering in progress on the TS feed should be stopped before calling this function. The **demux** function parameter contains a pointer to the demux API and instance data. The **feed** function parameter contains a pointer to the TS feed API and instance data. It returns: 0 on success; -EINVAL on bad parameter.

allocate_section_feed Allocates a new section feed, i.e. a demux resource for filtering and receiving sections. On platforms with hardware support for section filtering, a section feed is directly mapped to the demux HW. On other platforms, TS packets are first PID filtered in hardware and a hardware section filter then emulated in software. The caller obtains an API pointer of type dmx_section_feed_t as an out parameter. Using this API the caller can set filtering parameters and start receiving sections. The **demux** function

parameter contains a pointer to the demux API and instance data. The **feed** function parameter contains a pointer to the TS feed API and instance data. The **callback** function parameter contains a pointer to the callback function for passing received TS packet. It returns: 0 on success; -EBUSY, if no more TS feeds is available; -EINVAL, on bad parameter.

release_section_feed Releases the resources allocated with **allocate_section_feed**, including allocated filters. Any filtering in progress on the section feed should be stopped before calling this function. The **demux** function parameter contains a pointer to the demux API and instance data. The **feed** function parameter contains a pointer to the TS feed API and instance data. It returns: 0 on success; -EINVAL, on bad parameter.

add_frontend Registers a connectivity between a demux and a front-end, i.e., indicates that the demux can be connected via a call to **connect_frontend** to use the given front-end as a TS source. The client of this function has to allocate dynamic or static memory for the frontend structure and initialize its fields before calling this function. This function is normally called during the driver initialization. The caller must not free the memory of the frontend struct before successfully calling **remove_frontend**. The **demux** function parameter contains a pointer to the demux API and instance data. The **frontend** function parameter contains a pointer to the front-end instance data. It returns: 0 on success; -EINVAL, on bad parameter.

remove_frontend Indicates that the given front-end, registered by a call to **add_frontend**, can no longer be connected as a TS source by this demux. The function should be called when a front-end driver or a demux driver is removed from the system. If the front-end is in use, the function fails with the return value of -EBUSY. After successfully calling this function, the caller can free the memory of the frontend struct if it was dynamically allocated before the **add_frontend** operation. The **demux** function parameter contains a pointer to the demux API and instance data. The **frontend** function parameter contains a pointer to the front-end instance data. It returns: 0 on success; -ENODEV, if the front-end was not found, -EINVAL, on bad parameter.

get_frontends Provides the APIs of the front-ends that have been registered for this demux. Any of the front-ends obtained with this call can be used as a parameter for **connect_frontend**. The include file demux.h contains the macro `DMX_FE_ENTRY()` for converting an element of the generic type struct `list_head` * to the type struct `dmx_frontend` . The caller must not free the memory of any of the elements obtained via this function call. The **demux* function parameter contains a pointer to the demux API and instance data. It returns a struct list_head pointer to the list of front-end interfaces, or NULL in the case of an empty list.

connect_frontend Connects the TS output of the front-end to the input of the demux. A demux can only be connected to a front-end registered to the demux with the function **add_frontend**. It may or may not be possible to connect multiple demuxes to the same front-end, depending on the capabilities of the HW platform. When not used, the front-end should be released by calling **disconnect_frontend**. The **demux** function parameter contains a pointer to the demux API and instance data. The **frontend** function parameter contains a pointer to the front-end instance data. It returns: 0 on success; -EINVAL, on bad parameter.

disconnect_frontend Disconnects the demux and a front-end previously connected by a **connect_frontend** call. The **demux** function parameter contains a pointer to the demux API and instance data. It returns: 0 on success; -EINVAL on bad parameter.

get_pes_pids Get the PIDs for DMX_PES_AUDIO0, DMX_PES_VIDEO0, DMX_PES_TELETEXT0, DMX_PES_SUBTITLE0 and DMX_PES_PCR0. The **demux** function parameter contains a pointer to the demux API and instance data. The **pids** function parameter contains an array with five u16 elements where the PIDs will be stored. It returns: 0 on success; -EINVAL on bad parameter.

53.2.4 Digital TV Conditional Access kABI

struct **dvb_ca_en50221**
Structure describing a CA interface

Definition

```
struct dvb_ca_en50221 {
  struct module *owner;
  int (*read_attribute_mem)(struct dvb_ca_en50221 *ca, int slot, int␣
↪address);
  int (*write_attribute_mem)(struct dvb_ca_en50221 *ca, int slot, int␣
↪address, u8 value);
  int (*read_cam_control)(struct dvb_ca_en50221 *ca, int slot, u8 address);
  int (*write_cam_control)(struct dvb_ca_en50221 *ca, int slot, u8 address,
↪ u8 value);
  int (*read_data)(struct dvb_ca_en50221 *ca, int slot, u8 *ebuf, int␣
↪ecount);
  int (*write_data)(struct dvb_ca_en50221 *ca, int slot, u8 *ebuf, int␣
↪ecount);
  int (*slot_reset)(struct dvb_ca_en50221 *ca, int slot);
  int (*slot_shutdown)(struct dvb_ca_en50221 *ca, int slot);
  int (*slot_ts_enable)(struct dvb_ca_en50221 *ca, int slot);
  int (*poll_slot_status)(struct dvb_ca_en50221 *ca, int slot, int open);
  void *data;
  void *private;
};
```

Members

owner the module owning this structure

read_attribute_mem function for reading attribute memory on the CAM

write_attribute_mem function for writing attribute memory on the CAM

read_cam_control function for reading the control interface on the CAM

write_cam_control function for reading the control interface on the CAM

read_data function for reading data (block mode)

write_data function for writing data (block mode)

slot_reset function to reset the CAM slot

slot_shutdown function to shutdown a CAM slot

slot_ts_enable function to enable the Transport Stream on a CAM slot

poll_slot_status function to poll slot status. Only necessary if DVB_CA_FLAG_EN50221_IRQ_CAMCHANGE is not set.

data private data, used by caller.

private Opaque data used by the dvb_ca core. Do not modify!

NOTE

the read_*, write_* and poll_slot_status functions will be called for different slots concurrently and need to use locks where and if appropriate. There will be no concurrent access to one slot.

void **dvb_ca_en50221_camchange_irq**(struct dvb_ca_en50221 * pubca, int slot, int change_type)
A CAMCHANGE IRQ has occurred.

Parameters

struct dvb_ca_en50221 * pubca CA instance.

int slot Slot concerned.

int change_type One of the DVB_CA_CAMCHANGE_* values

void **dvb_ca_en50221_camready_irq**(struct dvb_ca_en50221 * pubca, int slot)
A CAMREADY IRQ has occurred.

Parameters

struct dvb_ca_en50221 * pubca CA instance.

int slot Slot concerned.

void **dvb_ca_en50221_frda_irq**(struct dvb_ca_en50221 * ca, int slot)
An FR or a DA IRQ has occurred.

Parameters

struct dvb_ca_en50221 * ca CA instance.

int slot Slot concerned.

int **dvb_ca_en50221_init**(struct dvb_adapter * dvb_adapter, struct dvb_ca_en50221 * ca, int flags, int slot_count)
Initialise a new DVB CA device.

Parameters

struct dvb_adapter * dvb_adapter DVB adapter to attach the new CA device to.

struct dvb_ca_en50221 * ca The dvb_ca instance.

int flags Flags describing the CA device (DVB_CA_EN50221_FLAG_*).

int slot_count Number of slots supported.

Description

return 0 on success, nonzero on failure

void **dvb_ca_en50221_release**(struct dvb_ca_en50221 * ca)
Release a DVB CA device.

Parameters

struct dvb_ca_en50221 * ca The associated dvb_ca instance.

53.2.5 Digital TV Network kABI

struct **dvb_net**
describes a DVB network interface

Definition

```
struct dvb_net {
  struct dvb_device *dvbdev;
  struct net_device *device[DVB_NET_DEVICES_MAX];
  int state[DVB_NET_DEVICES_MAX];
  unsigned int exit:1;
  struct dmx_demux *demux;
  struct mutex ioctl_mutex;
};
```

Members

dvbdev pointer to `struct dvb_device`.

device array of pointers to `struct net_device`.

state array of integers to each net device. A value different than zero means that the interface is in usage.

exit flag to indicate when the device is being removed.

demux pointer to `struct dmx_demux`.

ioctl_mutex protect access to this struct.

Description

Currently, the core supports up to `DVB_NET_DEVICES_MAX` (10) network devices.

int **dvb_net_init**(struct dvb_adapter * adap, struct dvb_net * dvbnet, struct dmx_demux * dmxdemux)
nitializes a digital TV network device and registers it.

Parameters

struct dvb_adapter * adap pointer to `struct dvb_adapter`.

struct dvb_net * dvbnet pointer to `struct dvb_net`.

struct dmx_demux * dmxdemux pointer to `struct dmx_demux`.

void **dvb_net_release**(struct dvb_net * dvbnet)
releases a digital TV network device and unregisters it.

Parameters

struct dvb_net * dvbnet pointer to `struct dvb_net`.

53.3 Remote Controller devices

53.3.1 Remote Controller core

The remote controller core implements infrastructure to receive and send remote controller keyboard keystrokes and mouse events.

Every time a key is pressed on a remote controller, a scan code is produced. Also, on most hardware, keeping a key pressed for more than a few dozens of milliseconds produce a repeat key event. That's somewhat similar to what a normal keyboard or mouse is handled internally on Linux[1]. So, the remote controller core is implemented on the top of the linux input/evdev interface.

However, most of the remote controllers use infrared (IR) to transmit signals. As there are several protocols used to modulate infrared signals, one important part of the core is dedicated to adjust the driver and the core system to support the infrared protocol used by the emitter.

The infrared transmission is done by blinking a infrared emitter using a carrier. The carrier can be switched on or off by the IR transmitter hardware. When the carrier is switched on, it is called PULSE. When the carrier is switched off, it is called SPACE.

In other words, a typical IR transmission can be viewed as a sequence of PULSE and SPACE events, each with a given duration.

The carrier parameters (frequency, duty cycle) and the intervals for PULSE and SPACE events depend on the protocol. For example, the NEC protocol uses a carrier of 38kHz, and transmissions start with a 9ms PULSE and a 4.5ms SPACE. It then transmits 16 bits of scan code, being 8 bits for address (usually it is a fixed number for a given remote controller), followed by 8 bits of code. A bit "1" is modulated with 560µs PULSE followed by 1690µs SPACE and a bit "0"is modulated with 560µs PULSE followed by 560µs SPACE.

At receiver, a simple low-pass filter can be used to convert the received signal in a sequence of PULSE/SPACE events, filtering out the carrier frequency. Due to that, the receiver doesn't care about the carrier's actual frequency parameters: all it has to do is to measure the amount of time it receives PULSE/SPACE events. So, a simple IR receiver hardware will just provide a sequence of timings for those events to the Kernel. The drivers for hardware with such kind of receivers are identified by `RC_DRIVER_IR_RAW`, as defined by `rc_driver_type`[2]. Other hardware come with a microcontroller that decode the PULSE/SPACE sequence and return scan codes to the Kernel. Such kind of receivers are identified by `RC_DRIVER_SCANCODE`.

When the RC core receives events produced by `RC_DRIVER_IR_RAW` IR receivers, it needs to decode the IR protocol, in order to obtain the corresponding scan code. The protocols supported by the RC core are defined at enum `rc_proto`.

[1] The main difference is that, on keyboard events, the keyboard controller produces one event for a key press and another one for key release. On infrared-based remote controllers, there's no key release event. Instead, an extra code is produced to indicate key repeats.

[2] The RC core also supports devices that have just IR emitters, without any receivers. Right now, all such devices work only in raw TX mode. Such kind of hardware is identified as `RC_DRIVER_IR_RAW_TX`.

When the RC code receives a scan code (either directly, by a driver of the type RC_DRIVER_SCANCODE, or via its IR decoders), it needs to convert into a Linux input event code. This is done via a mapping table.

The Kernel has support for mapping tables available on most media devices. It also supports loading a table in runtime, via some sysfs nodes. See the RC userspace API for more details.

Remote controller data structures and functions

enum **rc_driver_type**
type of the RC driver.

Constants

RC_DRIVER_SCANCODE Driver or hardware generates a scancode.

RC_DRIVER_IR_RAW Driver or hardware generates pulse/space sequences. It needs a Infra-Red pulse/space decoder

RC_DRIVER_IR_RAW_TX Device transmitter only, driver requires pulse/space data sequence.

struct **rc_scancode_filter**
Filter scan codes.

Definition

```
struct rc_scancode_filter {
  u32 data;
  u32 mask;
};
```

Members

data Scancode data to match.

mask Mask of bits of scancode to compare.

enum **rc_filter_type**
Filter type constants.

Constants

RC_FILTER_NORMAL Filter for normal operation.

RC_FILTER_WAKEUP Filter for waking from suspend.

RC_FILTER_MAX Number of filter types.

struct **lirc_fh**
represents an open lirc file

Definition

```
struct lirc_fh {
  struct list_head list;
  struct rc_dev *rc;
  int carrier_low;
```

(continues on next page)

(continued from previous page)

```
  bool send_timeout_reports;
  unsigned int *rawir;
  struct lirc_scancode *scancodes;
  wait_queue_head_t wait_poll;
  u8 send_mode;
  u8 rec_mode;
};
```

Members

list list of open file handles

rc rcdev for this lirc chardev

carrier_low when setting the carrier range, first the low end must be set with an ioctl and then the high end with another ioctl

send_timeout_reports report timeouts in lirc raw IR.

rawir queue for incoming raw IR

scancodes queue for incoming decoded scancodes

wait_poll poll struct for lirc device

send_mode lirc mode for sending, either LIRC_MODE_SCANCODE or LIRC_MODE_PULSE

rec_mode lirc mode for receiving, either LIRC_MODE_SCANCODE or LIRC_MODE_MODE2

struct **rc_dev**
represents a remote control device

Definition

```
struct rc_dev {
  struct device                   dev;
  bool managed_alloc;
  const struct attribute_group    *sysfs_groups[5];
  const char                      *device_name;
  const char                      *input_phys;
  struct input_id                 input_id;
  const char                      *driver_name;
  const char                      *map_name;
  struct rc_map                   rc_map;
  struct mutex                    lock;
  unsigned int                    minor;
  struct ir_raw_event_ctrl        *raw;
  struct input_dev                *input_dev;
  enum rc_driver_type             driver_type;
  bool idle;
  bool encode_wakeup;
  u64 allowed_protocols;
  u64 enabled_protocols;
  u64 allowed_wakeup_protocols;
  enum rc_proto                   wakeup_protocol;
  struct rc_scancode_filter       scancode_filter;
```

(continues on next page)

(continued from previous page)

```
  struct rc_scancode_filter       scancode_wakeup_filter;
  u32 scancode_mask;
  u32 users;
  void *priv;
  spinlock_t keylock;
  bool keypressed;
  unsigned long                   keyup_jiffies;
  struct timer_list               timer_keyup;
  struct timer_list               timer_repeat;
  u32 last_keycode;
  enum rc_proto                   last_protocol;
  u64 last_scancode;
  u8 last_toggle;
  u32 timeout;
  u32 min_timeout;
  u32 max_timeout;
  u32 rx_resolution;
  u32 tx_resolution;
#ifdef CONFIG_LIRC;
  struct device                   lirc_dev;
  struct cdev                     lirc_cdev;
  ktime_t gap_start;
  u64 gap_duration;
  bool gap;
  spinlock_t lirc_fh_lock;
  struct list_head                lirc_fh;
#endif;
  bool registered;
  int (*change_protocol)(struct rc_dev *dev, u64 *rc_proto);
  int (*open)(struct rc_dev *dev);
  void (*close)(struct rc_dev *dev);
  int (*s_tx_mask)(struct rc_dev *dev, u32 mask);
  int (*s_tx_carrier)(struct rc_dev *dev, u32 carrier);
  int (*s_tx_duty_cycle)(struct rc_dev *dev, u32 duty_cycle);
  int (*s_rx_carrier_range)(struct rc_dev *dev, u32 min, u32 max);
  int (*tx_ir)(struct rc_dev *dev, unsigned *txbuf, unsigned n);
  void (*s_idle)(struct rc_dev *dev, bool enable);
  int (*s_learning_mode)(struct rc_dev *dev, int enable);
  int (*s_carrier_report) (struct rc_dev *dev, int enable);
  int (*s_filter)(struct rc_dev *dev, struct rc_scancode_filter *filter);
  int (*s_wakeup_filter)(struct rc_dev *dev, struct rc_scancode_filter␣
↪*filter);
  int (*s_timeout)(struct rc_dev *dev, unsigned int timeout);
};
```

Members

dev driver model’s view of this device

managed_alloc devm_rc_allocate_device was used to create rc_dev

sysfs_groups sysfs attribute groups

device_name name of the rc child device

input_phys physical path to the input child device

input_id id of the input child device (struct input_id)

driver_name name of the hardware driver which registered this device

map_name name of the default keymap

rc_map current scan/key table

lock used to ensure we' ve filled in all protocol details before anyone can call show_protocols or store_protocols

minor unique minor remote control device number

raw additional data for raw pulse/space devices

input_dev the input child device used to communicate events to userspace

driver_type specifies if protocol decoding is done in hardware or software

idle used to keep track of RX state

encode_wakeup wakeup filtering uses IR encode API, therefore the allowed wakeup protocols is the set of all raw encoders

allowed_protocols bitmask with the supported RC_PROTO_BIT_* protocols

enabled_protocols bitmask with the enabled RC_PROTO_BIT_* protocols

allowed_wakeup_protocols bitmask with the supported RC_PROTO_BIT_* wakeup protocols

wakeup_protocol the enabled RC_PROTO_* wakeup protocol or RC_PROTO_UNKNOWN if disabled.

scancode_filter scancode filter

scancode_wakeup_filter scancode wakeup filters

scancode_mask some hardware decoders are not capable of providing the full scancode to the application. As this is a hardware limit, we can' t do anything with it. Yet, as the same keycode table can be used with other devices, a mask is provided to allow its usage. Drivers should generally leave this field in blank

users number of current users of the device

priv driver-specific data

keylock protects the remaining members of the struct

keypressed whether a key is currently pressed

keyup_jiffies time (in jiffies) when the current keypress should be released

timer_keyup timer for releasing a keypress

timer_repeat timer for autorepeat events. This is needed for CEC, which has non-standard repeats.

last_keycode keycode of last keypress

last_protocol protocol of last keypress

last_scancode scancode of last keypress

last_toggle toggle value of last command

timeout optional time after which device stops sending data

min_timeout minimum timeout supported by device

max_timeout maximum timeout supported by device

rx_resolution resolution (in ns) of input sampler

tx_resolution resolution (in ns) of output sampler

lirc_dev lirc device

lirc_cdev lirc char cdev

gap_start time when gap starts

gap_duration duration of initial gap

gap true if we' re in a gap

lirc_fh_lock protects lirc_fh list

lirc_fh list of open files

registered set to true by `rc_register_device()`, false by rc_unregister_device

change_protocol allow changing the protocol used on hardware decoders

open callback to allow drivers to enable polling/irq when IR input device is opened.

close callback to allow drivers to disable polling/irq when IR input device is opened.

s_tx_mask set transmitter mask (for devices with multiple tx outputs)

s_tx_carrier set transmit carrier frequency

s_tx_duty_cycle set transmit duty cycle (0% - 100%)

s_rx_carrier_range inform driver about carrier it is expected to handle

tx_ir transmit IR

s_idle enable/disable hardware idle mode, upon which, device doesn' t interrupt host until it sees IR pulses

s_learning_mode enable wide band receiver used for learning

s_carrier_report enable carrier reports

s_filter set the scancode filter

s_wakeup_filter set the wakeup scancode filter. If the mask is zero then wakeup should be disabled. wakeup_protocol will be set to a valid protocol if mask is nonzero.

s_timeout set hardware timeout in ns

struct rc_dev * **rc_allocate_device**(enum rc_driver_type)
Allocates a RC device

Parameters

enum rc_driver_type specifies the type of the RC output to be allocated returns a pointer to struct rc_dev.

struct rc_dev * **devm_rc_allocate_device**(struct device * dev, enum rc_driver_type)
Managed RC device allocation

Parameters

struct device * dev pointer to struct device

enum rc_driver_type specifies the type of the RC output to be allocated returns a pointer to struct rc_dev.

void **rc_free_device**(struct rc_dev * dev)
Frees a RC device

Parameters

struct rc_dev * dev pointer to struct rc_dev.

int **rc_register_device**(struct rc_dev * dev)
Registers a RC device

Parameters

struct rc_dev * dev pointer to struct rc_dev.

int **devm_rc_register_device**(struct device * parent, struct rc_dev * dev)
Manageded registering of a RC device

Parameters

struct device * parent pointer to struct device.

struct rc_dev * dev pointer to struct rc_dev.

void **rc_unregister_device**(struct rc_dev * dev)
Unregisters a RC device

Parameters

struct rc_dev * dev pointer to struct rc_dev.

struct **rc_map_table**
represents a scancode/keycode pair

Definition

```
struct rc_map_table {
  u64 scancode;
  u32 keycode;
};
```

Members

scancode scan code (u64)

keycode Linux input keycode

struct **rc_map**
represents a keycode map table

Definition

```
struct rc_map {
  struct rc_map_table     *scan;
  unsigned int            size;
  unsigned int            len;
  unsigned int            alloc;
  enum rc_proto           rc_proto;
  const char              *name;
  spinlock_t lock;
};
```

Members

scan pointer to struct rc_map_table

size Max number of entries

len Number of entries that are in use

alloc size of *scan, in bytes

rc_proto type of the remote controller protocol, as defined at enum rc_proto

name name of the key map table

lock lock to protect access to this structure

struct **rc_map_list**
list of the registered rc_map maps

Definition

```
struct rc_map_list {
  struct list_head         list;
  struct rc_map map;
};
```

Members

list pointer to struct list_head

map pointer to struct rc_map

int **rc_map_register**(struct rc_map_list * map)
Registers a Remote Controller scancode map

Parameters

struct rc_map_list * map pointer to struct rc_map_list

void **rc_map_unregister**(struct rc_map_list * map)
Unregisters a Remote Controller scancode map

Parameters

struct rc_map_list * map pointer to struct rc_map_list

struct rc_map * **rc_map_get**(const char * name)
gets an RC map from its name

Parameters

const char * name name of the RC scancode map

53.4 Media Controller devices

53.4.1 Media Controller

The media controller userspace API is documented in the Media Controller uAPI book. This document focus on the kernel-side implementation of the media framework.

Abstract media device model

Discovering a device internal topology, and configuring it at runtime, is one of the goals of the media framework. To achieve this, hardware devices are modelled as an oriented graph of building blocks called entities connected through pads.

An entity is a basic media hardware building block. It can correspond to a large variety of logical blocks such as physical hardware devices (CMOS sensor for instance), logical hardware devices (a building block in a System-on-Chip image processing pipeline), DMA channels or physical connectors.

A pad is a connection endpoint through which an entity can interact with other entities. Data (not restricted to video) produced by an entity flows from the entity' s output to one or more entity inputs. Pads should not be confused with physical pins at chip boundaries.

A link is a point-to-point oriented connection between two pads, either on the same entity or on different entities. Data flows from a source pad to a sink pad.

Media device

A media device is represented by a struct `media_device` instance, defined in `include/media/media-device.h`. Allocation of the structure is handled by the media device driver, usually by embedding the `media_device` instance in a larger driver-specific structure.

Drivers register media device instances by calling `__media_device_register()` via the macro `media_device_register()` and unregistered by calling `media_device_unregister()`.

Entities

Entities are represented by a struct `media_entity` instance, defined in `include/media/media-entity.h`. The structure is usually embedded into a higher-level structure, such as `v4l2_subdev` or `video_device` instances, although drivers can allocate entities directly.

Drivers initialize entity pads by calling `media_entity_pads_init()`.

Drivers register entities with a media device by calling `media_device_register_entity()` and unregistered by calling `media_device_unregister_entity()`.

Interfaces

Interfaces are represented by a struct `media_interface` instance, defined in `include/media/media-entity.h`. Currently, only one type of interface is defined: a device node. Such interfaces are represented by a struct `media_intf_devnode`.

Drivers initialize and create device node interfaces by calling `media_devnode_create()` and remove them by calling: `media_devnode_remove()`.

Pads

Pads are represented by a struct `media_pad` instance, defined in `include/media/media-entity.h`. Each entity stores its pads in a pads array managed by the entity driver. Drivers usually embed the array in a driver-specific structure.

Pads are identified by their entity and their 0-based index in the pads array.

Both information are stored in the struct `media_pad`, making the struct `media_pad` pointer the canonical way to store and pass link references.

Pads have flags that describe the pad capabilities and state.

`MEDIA_PAD_FL_SINK` indicates that the pad supports sinking data. `MEDIA_PAD_FL_SOURCE` indicates that the pad supports sourcing data.

Note: One and only one of `MEDIA_PAD_FL_SINK` or `MEDIA_PAD_FL_SOURCE` must be set for each pad.

Links

Links are represented by a struct `media_link` instance, defined in `include/media/media-entity.h`. There are two types of links:

1. pad to pad links:

Associate two entities via their PADs. Each entity has a list that points to all links originating at or targeting any of its pads. A given link is thus stored twice, once in the source entity and once in the target entity.

Drivers create pad to pad links by calling: `media_create_pad_link()` and remove with `media_entity_remove_links()`.

2. interface to entity links:

Associate one interface to a Link.

Drivers create interface to entity links by calling: `media_create_intf_link()` and remove with `media_remove_intf_links()`.

Note: Links can only be created after having both ends already created.

Links have flags that describe the link capabilities and state. The valid values are described at media_create_pad_link() and media_create_intf_link().

Graph traversal

The media framework provides APIs to iterate over entities in a graph.

To iterate over all entities belonging to a media device, drivers can use the media_device_for_each_entity macro, defined in include/media/media-device.h.

```
struct media_entity *entity;

media_device_for_each_entity(entity, mdev) {
// entity will point to each entity in turn
...
}
```

Drivers might also need to iterate over all entities in a graph that can be reached only through enabled links starting at a given entity. The media framework provides a depth-first graph traversal API for that purpose.

Note: Graphs with cycles (whether directed or undirected) are **NOT** supported by the graph traversal API. To prevent infinite loops, the graph traversal code limits the maximum depth to MEDIA_ENTITY_ENUM_MAX_DEPTH, currently defined as 16.

Drivers initiate a graph traversal by calling media_graph_walk_start()

The graph structure, provided by the caller, is initialized to start graph traversal at the given entity.

Drivers can then retrieve the next entity by calling media_graph_walk_next()

When the graph traversal is complete the function will return NULL.

Graph traversal can be interrupted at any moment. No cleanup function call is required and the graph structure can be freed normally.

Helper functions can be used to find a link between two given pads, or a pad connected to another pad through an enabled link media_entity_find_link() and media_entity_remote_pad().

Use count and power handling

Due to the wide differences between drivers regarding power management needs, the media controller does not implement power management. However, the struct media_entity includes a use_count field that media drivers can use to track the number of users of every entity for power management needs.

The media_entity.use_count field is owned by media drivers and must not be touched by entity drivers. Access to the field must be protected by the media_device.graph_mutex lock.

Links setup

Link properties can be modified at runtime by calling `media_entity_setup_link()`.

Pipelines and media streams

When starting streaming, drivers must notify all entities in the pipeline to prevent link states from being modified during streaming by calling `media_pipeline_start()`.

The function will mark all entities connected to the given entity through enabled links, either directly or indirectly, as streaming.

The struct `media_pipeline` instance pointed to by the pipe argument will be stored in every entity in the pipeline. Drivers should embed the struct `media_pipeline` in higher-level pipeline structures and can then access the pipeline through the struct `media_entity` pipe field.

Calls to `media_pipeline_start()` can be nested. The pipeline pointer must be identical for all nested calls to the function.

`media_pipeline_start()` may return an error. In that case, it will clean up any of the changes it did by itself.

When stopping the stream, drivers must notify the entities with `media_pipeline_stop()`.

If multiple calls to `media_pipeline_start()` have been made the same number of `media_pipeline_stop()` calls are required to stop streaming. The `media_entity.pipe` field is reset to `NULL` on the last nested stop call.

Link configuration will fail with `-EBUSY` by default if either end of the link is a streaming entity. Links that can be modified while streaming must be marked with the `MEDIA_LNK_FL_DYNAMIC` flag.

If other operations need to be disallowed on streaming entities (such as changing entities configuration parameters) drivers can explicitly check the media_entity stream_count field to find out if an entity is streaming. This operation must be done with the media_device graph_mutex held.

Link validation

Link validation is performed by `media_pipeline_start()` for any entity which has sink pads in the pipeline. The `media_entity.link_validate()` callback is used for that purpose. In `link_validate()` callback, entity driver should check that the properties of the source pad of the connected entity and its own sink pad match. It is up to the type of the entity (and in the end, the properties of the hardware) what matching actually means.

Subsystems should facilitate link validation by providing subsystem specific helper functions to provide easy access for commonly needed information, and in the end provide a way to use driver-specific callbacks.

Media Controller Device Allocator API

When the media device belongs to more than one driver, the shared media device is allocated with the shared struct device as the key for look ups.

The shared media device should stay in registered state until the last driver unregisters it. In addition, the media device should be released when all the references are released. Each driver gets a reference to the media device during probe, when it allocates the media device. If media device is already allocated, the allocate API bumps up the refcount and returns the existing media device. The driver puts the reference back in its disconnect routine when it calls `media_device_delete()`.

The media device is unregistered and cleaned up from the kref put handler to ensure that the media device stays in registered state until the last driver unregisters the media device.

Driver Usage

Drivers should use the appropriate media-core routines to manage the shared media device life-time handling the two states: 1. allocate -> register -> delete 2. get reference to already registered device -> delete

call `media_device_delete()` routine to make sure the shared media device delete is handled correctly.

driver probe: Call `media_device_usb_allocate()` to allocate or get a reference Call `media_device_register()`, if media devnode isn't registered

driver disconnect: Call `media_device_delete()` to free the media_device. Freeing is handled by the kref put handler.

API Definitions

struct **media_entity_notify**
Media Entity Notify

Definition

```
struct media_entity_notify {
  struct list_head list;
  void *notify_data;
  void (*notify)(struct media_entity *entity, void *notify_data);
};
```

Members

list List head

notify_data Input data to invoke the callback

notify Callback function pointer

Description

Drivers may register a callback to take action when new entities get registered with the media device. This handler is intended for creating links between existing entities and should not create entities and register them.

struct **media_device_ops**
Media device operations

Definition

```
struct media_device_ops {
  int (*link_notify)(struct media_link *link, u32 flags, unsigned int␣
↪notification);
  struct media_request *(*req_alloc)(struct media_device *mdev);
  void (*req_free)(struct media_request *req);
  int (*req_validate)(struct media_request *req);
  void (*req_queue)(struct media_request *req);
};
```

Members

link_notify Link state change notification callback. This callback is called with the graph_mutex held.

req_alloc Allocate a request. Set this if you need to allocate a struct larger then struct media_request. **req_alloc** and **req_free** must either both be set or both be NULL.

req_free Free a request. Set this if **req_alloc** was set as well, leave to NULL otherwise.

req_validate Validate a request, but do not queue yet. The req_queue_mutex lock is held when this op is called.

req_queue Queue a validated request, cannot fail. If something goes wrong when queueing this request then it should be marked as such internally in the driver and any related buffers must eventually return to vb2 with state VB2_BUF_STATE_ERROR. The req_queue_mutex lock is held when this op is called. It is important that vb2 buffer objects are queued last after all other object types are queued: queueing a buffer kickstarts the request processing, so all other objects related to the request (and thus the buffer) must be available to the driver. And once a buffer is queued, then the driver can complete or delete objects from the request before req_queue exits.

struct **media_device**
Media device

Definition

```
struct media_device {
  struct device *dev;
  struct media_devnode *devnode;
  char model[32];
  char driver_name[32];
  char serial[40];
  char bus_info[32];
  u32 hw_revision;
  u64 topology_version;
  u32 id;
  struct ida entity_internal_idx;
  int entity_internal_idx_max;
  struct list_head entities;
```

(continues on next page)

(continued from previous page)

```
  struct list_head interfaces;
  struct list_head pads;
  struct list_head links;
  struct list_head entity_notify;
  struct mutex graph_mutex;
  struct media_graph pm_count_walk;
  void *source_priv;
  int (*enable_source)(struct media_entity *entity, struct media_pipeline␣
↪*pipe);
  void (*disable_source)(struct media_entity *entity);
  const struct media_device_ops *ops;
  struct mutex req_queue_mutex;
  atomic_t request_id;
};
```

Members

dev Parent device

devnode Media device node

model Device model name

driver_name Optional device driver name. If not set, calls to `MEDIA_IOC_DEVICE_INFO` will return `dev->driver->name`. This is needed for USB drivers for example, as otherwise they' ll all appear as if the driver name was "usb" .

serial Device serial number (optional)

bus_info Unique and stable device location identifier

hw_revision Hardware device revision

topology_version Monotonic counter for storing the version of the graph topology. Should be incremented each time the topology changes.

id Unique ID used on the last registered graph object

entity_internal_idx Unique internal entity ID used by the graph traversal algorithms

entity_internal_idx_max Allocated internal entity indices

entities List of registered entities

interfaces List of registered interfaces

pads List of registered pads

links List of registered links

entity_notify List of registered entity_notify callbacks

graph_mutex Protects access to struct media_device data

pm_count_walk Graph walk for power state walk. Access serialised using graph_mutex.

source_priv Driver Private data for enable/disable source handlers

enable_source Enable Source Handler function pointer

disable_source Disable Source Handler function pointer

ops Operation handler callbacks

req_queue_mutex Serialise the MEDIA_REQUEST_IOC_QUEUE ioctl w.r.t. other operations that stop or start streaming.

request_id Used to generate unique request IDs

Description

This structure represents an abstract high-level media device. It allows easy access to entities and provides basic media device-level support. The structure can be allocated directly or embedded in a larger structure.

The parent **dev** is a physical device. It must be set before registering the media device.

model is a descriptive model name exported through sysfs. It doesn't have to be unique.

enable_source is a handler to find source entity for the sink entity and activate the link between them if source entity is free. Drivers should call this handler before accessing the source.

disable_source is a handler to find source entity for the sink entity and deactivate the link between them. Drivers should call this handler to release the source.

Use-case: find tuner entity connected to the decoder entity and check if it is available, and activate the the link between them from **enable_source** and deactivate from **disable_source**.

Note: Bridge driver is expected to implement and set the handler when `media_device` is registered or when bridge driver finds the media_device during probe. Bridge driver sets source_priv with information necessary to run **enable_source** and **disable_source** handlers. Callers should hold graph_mutex to access and call **enable_source** and **disable_source** handlers.

---

int **media_entity_enum_init**(struct media_entity_enum * ent_enum, struct media_device * mdev)
Initialise an entity enumeration

Parameters

struct media_entity_enum * ent_enum Entity enumeration to be initialised

struct media_device * mdev The related media device

Return

zero on success or a negative error code.

void **media_device_init**(struct media_device * mdev)
Initializes a media device element

Parameters

struct media_device * mdev pointer to struct `media_device`

Description

This function initializes the media device prior to its registration. The media device initialization and registration is split in two functions to avoid race conditions and make the media device available to user-space before the media graph has been completed.

So drivers need to first initialize the media device, register any entity within the media device, create pad to pad links and then finally register the media device by calling `media_device_register()` as a final step.

void **media_device_cleanup**(struct media_device * mdev)
Cleanups a media device element

Parameters

struct media_device * mdev pointer to struct `media_device`

Description

This function that will destroy the graph_mutex that is initialized in `media_device_init()`.

int **__media_device_register**(struct media_device * mdev, struct module * owner)
Registers a media device element

Parameters

struct media_device * mdev pointer to struct `media_device`

struct module * owner should be filled with `THIS_MODULE`

Description

Users, should, instead, call the `media_device_register()` macro.

The caller is responsible for initializing the `media_device` structure before registration. The following fields of `media_device` must be set:

- `media_entity.dev` must point to the parent device (usually a `pci_dev`, `usb_interface` or `platform_device` instance).
- `media_entity.model` must be filled with the device model name as a NUL-terminated UTF-8 string. The device/model revision must not be stored in this field.

The following fields are optional:

- `media_entity.serial` is a unique serial number stored as a NUL-terminated ASCII string. The field is big enough to store a GUID in text form. If the hardware doesn't provide a unique serial number this field must be left empty.
- `media_entity.bus_info` represents the location of the device in the system as a NUL-terminated ASCII string. For PCI/PCIe devices `media_entity.bus_info` must be set to "PCI:"(or "PCIe:") followed by the value of pci_name(). For USB devices,the `usb_make_path()` function must be used. This field is used by applications to distinguish between otherwise identical devices that don't provide a serial number.

- media_entity.hw_revision is the hardware device revision in a driver-specific format. When possible the revision should be formatted with the KERNEL_VERSION() macro.

Note:

1) Upon successful registration a character device named media[0-9]+ is created. The device major and minor numbers are dynamic. The model name is exported as a sysfs attribute.

2) Unregistering a media device that hasn't been registered is **NOT** safe.

Return

returns zero on success or a negative error code.

media_device_register(mdev)
Registers a media device element

Parameters

mdev pointer to struct media_device

Description

This macro calls __media_device_register() passing THIS_MODULE as the __media_device_register() second argument (**owner**).

void **media_device_unregister**(struct media_device * mdev)
Unregisters a media device element

Parameters

struct media_device * mdev pointer to struct media_device

Description

It is safe to call this function on an unregistered (but initialised) media device.

int **media_device_register_entity**(struct media_device * mdev, struct media_entity * entity)
registers a media entity inside a previously registered media device.

Parameters

struct media_device * mdev pointer to struct media_device

struct media_entity * entity pointer to struct media_entity to be registered

Description

Entities are identified by a unique positive integer ID. The media controller framework will such ID automatically. IDs are not guaranteed to be contiguous, and the ID number can change on newer Kernel versions. So, neither the driver nor userspace should hardcode ID numbers to refer to the entities, but, instead, use the framework to find the ID, when needed.

The media_entity name, type and flags fields should be initialized before calling media_device_register_entity(). Entities embedded in higher-level standard structures can have some of those fields set by the higher-level framework.

If the device has pads, `media_entity_pads_init()` should be called before this function. Otherwise, the `media_entity.pad` and `media_entity.num_pads` should be zeroed before calling this function.

Entities have flags that describe the entity capabilities and state:

`MEDIA_ENT_FL_DEFAULT` indicates the default entity for a given type. This can be used to report the default audio and video devices or the default camera sensor.

Note: Drivers should set the entity function before calling this function. Please notice that the values `MEDIA_ENT_F_V4L2_SUBDEV_UNKNOWN` and `MEDIA_ENT_F_UNKNOWN` should not be used by the drivers.

---

void **`media_device_unregister_entity`**(struct media_entity * entity)
unregisters a media entity.

Parameters

`struct media_entity * entity` pointer to struct `media_entity` to be unregistered

Description

All links associated with the entity and all PADs are automatically unregistered from the media_device when this function is called.

Unregistering an entity will not change the IDs of the other entities and the previoully used ID will never be reused for a newly registered entities.

When a media device is unregistered, all its entities are unregistered automatically. No manual entities unregistration is then required.

Note: The media_entity instance itself must be freed explicitly by the driver if required.

---

int **`media_device_register_entity_notify`**(struct media_device * mdev, struct media_entity_notify * nptr)
Registers a media entity_notify callback

Parameters

`struct media_device * mdev` The media device

`struct media_entity_notify * nptr` The media_entity_notify

Description

Note: When a new entity is registered, all the registered media_entity_notify callbacks are invoked.

---

void **media_device_unregister_entity_notify**(struct media_device * mdev, struct media_entity_notify * nptr)
Unregister a media entity notify callback

Parameters

struct media_device * mdev The media device

struct media_entity_notify * nptr The media_entity_notify

void **media_device_pci_init**(struct media_device * mdev, struct pci_dev * pci_dev, const char * name)
create and initialize a struct `media_device` from a PCI device.

Parameters

struct media_device * mdev pointer to struct `media_device`

struct pci_dev * pci_dev pointer to struct pci_dev

const char * name media device name. If `NULL`, the routine will use the default name for the pci device, given by pci_name() macro.

void **__media_device_usb_init**(struct media_device * mdev, struct usb_device * udev, const char * board_name, const char * driver_name)
create and initialize a struct `media_device` from a PCI device.

Parameters

struct media_device * mdev pointer to struct `media_device`

struct usb_device * udev pointer to struct usb_device

const char * board_name media device name. If `NULL`, the routine will use the usb product name, if available.

const char * driver_name name of the driver. if `NULL`, the routine will use the name given by `udev->dev->driver->name`, with is usually the wrong thing to do.

Description

Note: It is better to call `media_device_usb_init()` instead, as such macro fills driver_name with `KBUILD_MODNAME`.

---

media_device_usb_init(mdev, udev, name)
create and initialize a struct `media_device` from a PCI device.

Parameters

mdev pointer to struct `media_device`

udev pointer to struct usb_device

name media device name. If `NULL`, the routine will use the usb product name, if available.

Description

This macro calls `media_device_usb_init()` passing the `media_device_usb_init()` **driver_name** parameter filled with `KBUILD_MODNAME`.

struct **media_file_operations**
Media device file operations

Definition

```
struct media_file_operations {
  struct module *owner;
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  __poll_t (*poll) (struct file *, struct poll_table_struct *);
  long (*ioctl) (struct file *, unsigned int, unsigned long);
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  int (*open) (struct file *);
  int (*release) (struct file *);
};
```

Members

owner should be filled with `THIS_MODULE`

read pointer to the function that implements read() syscall

write pointer to the function that implements write() syscall

poll pointer to the function that implements poll() syscall

ioctl pointer to the function that implements ioctl() syscall

compat_ioctl pointer to the function that will handle 32 bits userspace calls to the the ioctl() syscall on a Kernel compiled with 64 bits.

open pointer to the function that implements open() syscall

release pointer to the function that will release the resources allocated by the **open** function.

struct **media_devnode**
Media device node

Definition

```
struct media_devnode {
  struct media_device *media_dev;
  const struct media_file_operations *fops;
  struct device dev;
  struct cdev cdev;
  struct device *parent;
  int minor;
  unsigned long flags;
  void (*release)(struct media_devnode *devnode);
};
```

Members

media_dev pointer to struct `media_device`

fops pointer to struct `media_file_operations` with media device ops

dev pointer to struct `device` containing the media controller device

cdev struct cdev pointer character device

parent parent device

minor device node minor number

flags flags, combination of the MEDIA_FLAG_* constants

release release callback called at the end of media_devnode_release() routine at media-device.c.

Description

This structure represents a media-related device node.

The **parent** is a physical device. It must be set by core or device drivers before registering the node.

int **media_devnode_register**(struct media_device * mdev, struct media_devnode * devnode, struct module * owner)
register a media device node

Parameters

struct media_device * mdev struct media_device we want to register a device node

struct media_devnode * devnode media device node structure we want to register

struct module * owner should be filled with THIS_MODULE

Description

The registration code assigns minor numbers and registers the new device node with the kernel. An error is returned if no free minor number can be found, or if the registration of the device node fails.

Zero is returned on success.

Note that if the media_devnode_register call fails, the release() callback of the media_devnode structure is not called, so the caller is responsible for freeing any data.

void **media_devnode_unregister_prepare**(struct media_devnode * devnode)
clear the media device node register bit

Parameters

struct media_devnode * devnode the device node to prepare for unregister

Description

This clears the passed device register bit. Future open calls will be met with errors. Should be called before media_devnode_unregister() to avoid races with unregister and device file open calls.

This function can safely be called if the device node has never been registered or has already been unregistered.

void **media_devnode_unregister**(struct media_devnode * devnode)
unregister a media device node

Parameters

struct media_devnode * devnode the device node to unregister

Description

This unregisters the passed device. Future open calls will be met with errors.

Should be called after `media_devnode_unregister_prepare()`

struct media_devnode * **media_devnode_data**(struct file * filp)
returns a pointer to the `media_devnode`

Parameters

struct file * filp pointer to struct `file`

int **media_devnode_is_registered**(struct media_devnode * devnode)
returns true if `media_devnode` is registered; false otherwise.

Parameters

struct media_devnode * devnode pointer to struct `media_devnode`.

Note

If mdev is NULL, it also returns false.

enum **media_gobj_type**
type of a graph object

Constants

MEDIA_GRAPH_ENTITY Identify a media entity

MEDIA_GRAPH_PAD Identify a media pad

MEDIA_GRAPH_LINK Identify a media link

MEDIA_GRAPH_INTF_DEVNODE Identify a media Kernel API interface via a device node

struct **media_gobj**
Define a graph object.

Definition

```
struct media_gobj {
  struct media_device     *mdev;
  u32 id;
  struct list_head        list;
};
```

Members

mdev Pointer to the struct `media_device` that owns the object

id Non-zero object ID identifier. The ID should be unique inside a media_device, as it is composed by `MEDIA_BITS_PER_TYPE` to store the type plus `MEDIA_BITS_PER_ID` to store the ID

list List entry stored in one of the per-type mdev object lists

Description

All objects on the media graph should have this struct embedded

struct **media_entity_enum**
An enumeration of media entities.

Definition

```
struct media_entity_enum {
  unsigned long *bmap;
  int idx_max;
};
```

Members

bmap Bit map in which each bit represents one entity at struct media_entity->internal_idx.

idx_max Number of bits in bmap

struct **media_graph**
Media graph traversal state

Definition

```
struct media_graph {
  struct {
    struct media_entity *entity;
    struct list_head *link;
  } stack[MEDIA_ENTITY_ENUM_MAX_DEPTH];
  struct media_entity_enum ent_enum;
  int top;
};
```

Members

stack Graph traversal stack; the stack contains information on the path the media entities to be walked and the links through which they were reached.

stack.entity pointer to `struct media_entity` at the graph.

stack.link pointer to `struct list_head`.

ent_enum Visited entities

top The top of the stack

struct **media_pipeline**
Media pipeline related information

Definition

```
struct media_pipeline {
  int streaming_count;
  struct media_graph graph;
};
```

Members

streaming_count Streaming start count - streaming stop count

graph Media graph walk during pipeline start / stop

struct **media_link**
A link object part of a media graph.

Definition

```
struct media_link {
  struct media_gobj graph_obj;
  struct list_head list;
  union {
    struct media_gobj *gobj0;
    struct media_pad *source;
    struct media_interface *intf;
  };
  union {
    struct media_gobj *gobj1;
    struct media_pad *sink;
    struct media_entity *entity;
  };
  struct media_link *reverse;
  unsigned long flags;
  bool is_backlink;
};
```

Members

graph_obj Embedded structure containing the media object common data

list Linked list associated with an entity or an interface that owns the link.

{unnamed_union} anonymous

gobj0 Part of a union. Used to get the pointer for the first graph_object of the link.

source Part of a union. Used only if the first object (gobj0) is a pad. In that case, it represents the source pad.

intf Part of a union. Used only if the first object (gobj0) is an interface.

{unnamed_union} anonymous

gobj1 Part of a union. Used to get the pointer for the second graph_object of the link.

sink Part of a union. Used only if the second object (gobj1) is a pad. In that case, it represents the sink pad.

entity Part of a union. Used only if the second object (gobj1) is an entity.

reverse Pointer to the link for the reverse direction of a pad to pad link.

flags Link flags, as defined in uapi/media.h (MEDIA_LNK_FL_*)

is_backlink Indicate if the link is a backlink.

enum **media_pad_signal_type**
type of the signal inside a media pad

Constants

PAD_SIGNAL_DEFAULT Default signal. Use this when all inputs or all outputs are uniquely identified by the pad number.

PAD_SIGNAL_ANALOG The pad contains an analog signal. It can be Radio Frequency, Intermediate Frequency, a baseband signal or sub-cariers. Tuner inputs, IF-PLL demodulators, composite and s-video signals should use it.

PAD_SIGNAL_DV Contains a digital video signal, with can be a bitstream of samples taken from an analog TV video source. On such case, it usually contains the VBI data on it.

PAD_SIGNAL_AUDIO Contains an Intermediate Frequency analog signal from an audio sub-carrier or an audio bitstream. IF signals are provided by tuners and consumed by audio AM/FM decoders. Bitstream audio is provided by an audio decoder.

struct **media_pad**
A media pad graph object.

Definition

```
struct media_pad {
  struct media_gobj graph_obj;
  struct media_entity *entity;
  u16 index;
  enum media_pad_signal_type sig_type;
  unsigned long flags;
};
```

Members

graph_obj Embedded structure containing the media object common data

entity Entity this pad belongs to

index Pad index in the entity pads array, numbered from 0 to n

sig_type Type of the signal inside a media pad

flags Pad flags, as defined in include/uapi/linux/media.h (seek for `MEDIA_PAD_FL_*`)

struct **media_entity_operations**
Media entity operations

Definition

```
struct media_entity_operations {
  int (*get_fwnode_pad)(struct media_entity *entity, struct fwnode_
↪endpoint *endpoint);
  int (*link_setup)(struct media_entity *entity,const struct media_pad␣
↪*local, const struct media_pad *remote, u32 flags);
  int (*link_validate)(struct media_link *link);
};
```

Members

get_fwnode_pad Return the pad number based on a fwnode endpoint or a negative value on error. This operation can be used to map a fwnode to a media pad number. Optional.

link_setup Notify the entity of link changes. The operation can return an error, in which case link setup will be cancelled. Optional.

link_validate Return whether a link is valid from the entity point of view. The `media_pipeline_start()` function validates all links by calling this operation. Optional.

Description

Note: Those these callbacks are called with struct `media_device.graph_mutex` mutex held.

---

enum **media_entity_type**
Media entity type

Constants

MEDIA_ENTITY_TYPE_BASE The entity isn't embedded in another subsystem structure.

MEDIA_ENTITY_TYPE_VIDEO_DEVICE The entity is embedded in a struct video_device instance.

MEDIA_ENTITY_TYPE_V4L2_SUBDEV The entity is embedded in a struct v4l2_subdev instance.

Description

Media entity objects are often not instantiated directly, but the media entity structure is inherited by (through embedding) other subsystem-specific structures. The media entity type identifies the type of the subclass structure that implements a media entity instance.

This allows runtime type identification of media entities and safe casting to the correct object type. For instance, a media entity structure instance embedded in a v4l2_subdev structure instance will have the type `MEDIA_ENTITY_TYPE_V4L2_SUBDEV` and can safely be cast to a `v4l2_subdev` structure using the `container_of()` macro.

struct **media_entity**
A media entity graph object.

Definition

```
struct media_entity {
  struct media_gobj graph_obj;
  const char *name;
  enum media_entity_type obj_type;
  u32 function;
  unsigned long flags;
  u16 num_pads;
  u16 num_links;
  u16 num_backlinks;
```

(continues on next page)

(continued from previous page)

```
  int internal_idx;
  struct media_pad *pads;
  struct list_head links;
  const struct media_entity_operations *ops;
  int stream_count;
  int use_count;
  struct media_pipeline *pipe;
  union {
    struct {
      u32 major;
      u32 minor;
    } dev;
  } info;
};
```

Members

graph_obj Embedded structure containing the media object common data.

name Entity name.

obj_type Type of the object that implements the media_entity.

function Entity main function, as defined in include/uapi/linux/media.h (seek for MEDIA_ENT_F_*)

flags Entity flags, as defined in include/uapi/linux/media.h (seek for MEDIA_ENT_FL_*)

num_pads Number of sink and source pads.

num_links Total number of links, forward and back, enabled and disabled.

num_backlinks Number of backlinks

internal_idx An unique internal entity specific number. The numbers are re-used if entities are unregistered or registered again.

pads Pads array with the size defined by **num_pads**.

links List of data links.

ops Entity operations.

stream_count Stream count for the entity.

use_count Use count for the entity.

pipe Pipeline this entity belongs to.

info Union with devnode information. Kept just for backward compatibility.

info.dev Contains device major and minor info.

info.dev.major device node major, if the device is a devnode.

info.dev.minor device node minor, if the device is a devnode.

Description

Note: **stream_count** and **use_count** reference counts must never be negative, but are signed integers on purpose: a simple `WARN_ON(<0)` check can be used to detect reference count bugs that would make them negative.

struct **media_interface**
A media interface graph object.

Definition

```
struct media_interface {
  struct media_gobj               graph_obj;
  struct list_head                links;
  u32 type;
  u32 flags;
};
```

Members

graph_obj embedded graph object

links List of links pointing to graph entities

type Type of the interface as defined in include/uapi/linux/media.h (seek for `MEDIA_INTF_T_*`)

flags Interface flags as defined in include/uapi/linux/media.h (seek for `MEDIA_INTF_FL_*`)

Description

Note: Currently, no flags for `media_interface` is defined.

struct **media_intf_devnode**
A media interface via a device node.

Definition

```
struct media_intf_devnode {
  struct media_interface          intf;
  u32 major;
  u32 minor;
};
```

Members

intf embedded interface object

major Major number of a device node

minor Minor number of a device node

u32 **media_entity_id**(struct media_entity * entity)
return the media entity graph object id

Parameters

struct media_entity * entity pointer to `media_entity`

enum media_gobj_type **media_type**(struct media_gobj * gobj)
return the media object type

Parameters

struct media_gobj * gobj Pointer to the struct `media_gobj` graph object

u32 **media_id**(struct media_gobj * gobj)
return the media object ID

Parameters

struct media_gobj * gobj Pointer to the struct `media_gobj` graph object

u32 **media_gobj_gen_id**(enum media_gobj_type type, u64 local_id)
encapsulates type and ID on at the object ID

Parameters

enum media_gobj_type type object type as define at enum `media_gobj_type`.

u64 local_id next ID, from struct `media_device.id`.

bool **is_media_entity_v4l2_video_device**(struct media_entity * entity)
Check if the entity is a video_device

Parameters

struct media_entity * entity pointer to entity

Return

`true` if the entity is an instance of a video_device object and can safely be cast to a struct video_device using the `container_of()` macro, or `false` otherwise.

bool **is_media_entity_v4l2_subdev**(struct media_entity * entity)
Check if the entity is a v4l2_subdev

Parameters

struct media_entity * entity pointer to entity

Return

`true` if the entity is an instance of a `v4l2_subdev` object and can safely be cast to a struct `v4l2_subdev` using the `container_of()` macro, or `false` otherwise.

int **__media_entity_enum_init**(struct media_entity_enum * ent_enum, int idx_max)
Initialise an entity enumeration

Parameters

struct media_entity_enum * ent_enum Entity enumeration to be initialised

int idx_max Maximum number of entities in the enumeration

Return

Returns zero on success or a negative error code.

void **media_entity_enum_cleanup**(struct media_entity_enum * ent_enum)
Release resources of an entity enumeration

Parameters

struct media_entity_enum * ent_enum Entity enumeration to be released

void **media_entity_enum_zero**(struct media_entity_enum * ent_enum)
Clear the entire enum

Parameters

struct media_entity_enum * ent_enum Entity enumeration to be cleared

void **media_entity_enum_set**(struct media_entity_enum * ent_enum, struct media_entity * entity)
Mark a single entity in the enum

Parameters

struct media_entity_enum * ent_enum Entity enumeration

struct media_entity * entity Entity to be marked

void **media_entity_enum_clear**(struct media_entity_enum * ent_enum, struct media_entity * entity)
Unmark a single entity in the enum

Parameters

struct media_entity_enum * ent_enum Entity enumeration

struct media_entity * entity Entity to be unmarked

bool **media_entity_enum_test**(struct media_entity_enum * ent_enum, struct media_entity * entity)
Test whether the entity is marked

Parameters

struct media_entity_enum * ent_enum Entity enumeration

struct media_entity * entity Entity to be tested

Description

Returns `true` if the entity was marked.

bool **media_entity_enum_test_and_set**(struct media_entity_enum * ent_enum, struct media_entity * entity)
Test whether the entity is marked, and mark it

Parameters

struct media_entity_enum * ent_enum Entity enumeration

struct media_entity * entity Entity to be tested

Description

Returns `true` if the entity was marked, and mark it before doing so.

bool **media_entity_enum_empty**(struct media_entity_enum * ent_enum)
Test whether the entire enum is empty

Parameters

struct media_entity_enum * ent_enum Entity enumeration

Return

`true` if the entity was empty.

bool **media_entity_enum_intersects**(struct media_entity_enum * ent_enum1, struct media_entity_enum * ent_enum2)
Test whether two enums intersect

Parameters

struct media_entity_enum * ent_enum1 First entity enumeration

struct media_entity_enum * ent_enum2 Second entity enumeration

Return

`true` if entity enumerations **ent_enum1** and **ent_enum2** intersect, otherwise `false`.

gobj_to_entity(gobj)
returns the struct `media_entity` pointer from the **gobj** contained on it.

Parameters

gobj Pointer to the struct `media_gobj` graph object

gobj_to_pad(gobj)
returns the struct `media_pad` pointer from the **gobj** contained on it.

Parameters

gobj Pointer to the struct `media_gobj` graph object

gobj_to_link(gobj)
returns the struct `media_link` pointer from the **gobj** contained on it.

Parameters

gobj Pointer to the struct `media_gobj` graph object

gobj_to_intf(gobj)
returns the struct `media_interface` pointer from the **gobj** contained on it.

Parameters

gobj Pointer to the struct `media_gobj` graph object

intf_to_devnode(intf)
returns the struct media_intf_devnode pointer from the **intf** contained on it.

Parameters

intf Pointer to struct `media_intf_devnode`

void **media_gobj_create**(struct media_device * mdev, enum media_gobj_type type, struct media_gobj * gobj)
Initialize a graph object

Parameters

struct media_device * mdev Pointer to the `media_device` that contains the object

enum media_gobj_type type Type of the object

struct media_gobj * gobj Pointer to the struct media_gobj graph object

Description

This routine initializes the embedded struct media_gobj inside a media graph object. It is called automatically if media_*_create function calls are used. However, if the object (entity, link, pad, interface) is embedded on some other object, this function should be called before registering the object at the media controller.

void **media_gobj_destroy**(struct media_gobj * gobj)
Stop using a graph object on a media device

Parameters

struct media_gobj * gobj Pointer to the struct media_gobj graph object

Description

This should be called by all routines like media_device_unregister() that remove/destroy media graph objects.

int **media_entity_pads_init**(struct media_entity * entity, u16 num_pads, struct media_pad * pads)
Initialize the entity pads

Parameters

struct media_entity * entity entity where the pads belong

u16 num_pads total number of sink and source pads

struct media_pad * pads Array of **num_pads** pads.

Description

The pads array is managed by the entity driver and passed to media_entity_pads_init() where its pointer will be stored in the media_entity structure.

If no pads are needed, drivers could either directly fill media_entity->num_pads with 0 and media_entity->pads with NULL or call this function that will do the same.

As the number of pads is known in advance, the pads array is not allocated dynamically but is managed by the entity driver. Most drivers will embed the pads array in a driver-specific structure, avoiding dynamic allocation.

Drivers must set the direction of every pad in the pads array before calling media_entity_pads_init(). The function will initialize the other pads fields.

void **media_entity_cleanup**(struct media_entity * entity)
free resources associated with an entity

Parameters

struct media_entity * entity entity where the pads belong

Description

This function must be called during the cleanup phase after unregistering the entity (currently, it does nothing).

int **media_get_pad_index**(struct media_entity * entity, bool is_sink, enum media_pad_signal_type sig_type)
retrieves a pad index from an entity

Parameters

struct media_entity * entity entity where the pads belong

bool is_sink true if the pad is a sink, false if it is a source

enum media_pad_signal_type sig_type type of signal of the pad to be search

Description

This helper function finds the first pad index inside an entity that satisfies both **is_sink** and **sig_type** conditions.

On success, return the pad number. If the pad was not found or the media entity is a NULL pointer, return -EINVAL.

Return

int **media_create_pad_link**(struct media_entity * source, u16 source_pad, struct media_entity * sink, u16 sink_pad, u32 flags)
creates a link between two entities.

Parameters

struct media_entity * source pointer to `media_entity` of the source pad.

u16 source_pad number of the source pad in the pads array

struct media_entity * sink pointer to `media_entity` of the sink pad.

u16 sink_pad number of the sink pad in the pads array.

u32 flags Link flags, as defined in include/uapi/linux/media.h (seek for `MEDIA_LNK_FL_*`)

Description

Valid values for flags:

MEDIA_LNK_FL_ENABLED Indicates that the link is enabled and can be used to transfer media data. When two or more links target a sink pad, only one of them can be enabled at a time.

MEDIA_LNK_FL_IMMUTABLE Indicates that the link enabled state can't be modified at runtime. If `MEDIA_LNK_FL_IMMUTABLE` is set, then `MEDIA_LNK_FL_ENABLED` must also be set, since an immutable link is always enabled.

Note: Before calling this function, `media_entity_pads_init()` and `media_device_register_entity()` should be called previously for both ends.

---

int **media_create_pad_links**(const struct media_device * mdev, const u32 source_function, struct media_entity * source, const u16 source_pad, const u32 sink_function, struct media_entity * sink, const u16 sink_pad, u32 flags, const bool allow_both_undefined)

creates a link between two entities.

Parameters

const struct media_device * mdev Pointer to the media_device that contains the object

const u32 source_function Function of the source entities. Used only if **source** is NULL.

struct media_entity * source pointer to `media_entity` of the source pad. If NULL, it will use all entities that matches the **sink_function**.

const u16 source_pad number of the source pad in the pads array

const u32 sink_function Function of the sink entities. Used only if **sink** is NULL.

struct media_entity * sink pointer to `media_entity` of the sink pad. If NULL, it will use all entities that matches the **sink_function**.

const u16 sink_pad number of the sink pad in the pads array.

u32 flags Link flags, as defined in include/uapi/linux/media.h.

const bool allow_both_undefined if `true`, then both **source** and **sink** can be NULL. In such case, it will create a crossbar between all entities that matches **source_function** to all entities that matches **sink_function**. If `false`, it will return 0 and won't create any link if both **source** and **sink** are NULL.

Description

Valid values for flags:

A `MEDIA_LNK_FL_ENABLED` flag indicates that the link is enabled and can be used to transfer media data. If multiple links are created and this flag is passed as an argument, only the first created link will have this flag.

A `MEDIA_LNK_FL_IMMUTABLE` flag indicates that the link enabled state can't be modified at runtime. If `MEDIA_LNK_FL_IMMUTABLE` is set, then `MEDIA_LNK_FL_ENABLED` must also be set since an immutable link is always enabled.

It is common for some devices to have multiple source and/or sink entities of the same type that should be linked. While `media_create_pad_link()` creates link by link, this function is meant to allow 1:n, n:1 and even cross-bar (n:n) links.

Note: Before calling this function, `media_entity_pads_init()` and `media_device_register_entity()` should be called previously for the entities to be linked.

---

void **media_entity_remove_links**(struct media_entity * entity)
remove all links associated with an entity

Parameters

struct media_entity * entity pointer to media_entity

Description

Note: This is called automatically when an entity is unregistered via media_device_register_entity().

---

int **__media_entity_setup_link**(struct media_link * link, u32 flags)
Configure a media link without locking

Parameters

struct media_link * link The link being configured

u32 flags Link configuration flags

Description

The bulk of link setup is handled by the two entities connected through the link. This function notifies both entities of the link configuration change.

If the link is immutable or if the current and new configuration are identical, return immediately.

The user is expected to hold link->source->parent->mutex. If not, media_entity_setup_link() should be used instead.

int **media_entity_setup_link**(struct media_link * link, u32 flags)
changes the link flags properties in runtime

Parameters

struct media_link * link pointer to media_link

u32 flags the requested new link flags

Description

The only configurable property is the MEDIA_LNK_FL_ENABLED link flag flag to enable/disable a link. Links marked with the MEDIA_LNK_FL_IMMUTABLE link flag can not be enabled or disabled.

When a link is enabled or disabled, the media framework calls the link_setup operation for the two entities at the source and sink of the link, in that order. If the second link_setup call fails, another link_setup call is made on the first entity to restore the original link flags.

Media device drivers can be notified of link setup operations by setting the media_device.link_notify pointer to a callback function. If provided, the notification callback will be called before enabling and after disabling links.

Entity drivers must implement the link_setup operation if any of their links is non-immutable. The operation must either configure the hardware or store the configuration information to be applied later.

Link configuration must not have any side effect on other links. If an enabled link at a sink pad prevents another link at the same pad from being enabled, the link_setup operation must return `-EBUSY` and can't implicitly disable the first enabled link.

Note: The valid values of the flags for the link is the same as described on `media_create_pad_link()`, for pad to pad links or the same as described on `media_create_intf_link()`, for interface to entity links.

---

struct media_link * **media_entity_find_link**(struct media_pad * source, struct media_pad * sink)

Find a link between two pads

Parameters

struct media_pad * source Source pad

struct media_pad * sink Sink pad

Return

returns a pointer to the link between the two entities. If no such link exists, return `NULL`.

struct media_pad * **media_entity_remote_pad**(const struct media_pad * pad)

Find the pad at the remote end of a link

Parameters

const struct media_pad * pad Pad at the local end of the link

Description

Search for a remote pad connected to the given pad by iterating over all links originating or terminating at that pad until an enabled link is found.

Return

returns a pointer to the pad at the remote end of the first found enabled link, or `NULL` if no enabled link has been found.

int **media_entity_get_fwnode_pad**(struct media_entity * entity, struct fwnode_handle * fwnode, unsigned long direction_flags)

Get pad number from fwnode

Parameters

struct media_entity * entity The entity

struct fwnode_handle * fwnode Pointer to the fwnode_handle which should be used to find the pad

unsigned long direction_flags Expected direction of the pad, as defined in include/uapi/linux/media.h (seek for `MEDIA_PAD_FL_*`)

Description

This function can be used to resolve the media pad number from a fwnode. This is useful for devices which use more complex mappings of media pads.

If the entity does not implement the get_fwnode_pad() operation then this function searches the entity for the first pad that matches the **direction_flags**.

Return

returns the pad number on success or a negative error code.

int **media_graph_walk_init**(struct media_graph * graph, struct media_device * mdev)
Allocate resources used by graph walk.

Parameters

struct media_graph * graph Media graph structure that will be used to walk the graph

struct media_device * mdev Pointer to the `media_device` that contains the object

void **media_graph_walk_cleanup**(struct media_graph * graph)
Release resources used by graph walk.

Parameters

struct media_graph * graph Media graph structure that will be used to walk the graph

void **media_graph_walk_start**(struct media_graph * graph, struct media_entity * entity)
Start walking the media graph at a given entity

Parameters

struct media_graph * graph Media graph structure that will be used to walk the graph

struct media_entity * entity Starting entity

Description

Before using this function, `media_graph_walk_init()` must be used to allocate resources used for walking the graph. This function initializes the graph traversal structure to walk the entities graph starting at the given entity. The traversal structure must not be modified by the caller during graph traversal. After the graph walk, the resources must be released using `media_graph_walk_cleanup()`.

struct media_entity * **media_graph_walk_next**(struct media_graph * graph)
Get the next entity in the graph

Parameters

struct media_graph * graph Media graph structure

Description

Perform a depth-first traversal of the given media entities graph.

The graph structure must have been previously initialized with a call to `media_graph_walk_start()`.

Return

returns the next entity in the graph or `NULL` if the whole graph have been traversed.

int **media_pipeline_start**(struct media_entity * entity, struct media_pipeline * pipe)
Mark a pipeline as streaming

Parameters

struct media_entity * entity Starting entity

struct media_pipeline * pipe Media pipeline to be assigned to all entities in the pipeline.

Description

Mark all entities connected to a given entity through enabled links, either directly or indirectly, as streaming. The given pipeline object is assigned to every entity in the pipeline and stored in the media_entity pipe field.

Calls to this function can be nested, in which case the same number of `media_pipeline_stop()` calls will be required to stop streaming. The pipeline pointer must be identical for all nested calls to `media_pipeline_start()`.

int **__media_pipeline_start**(struct media_entity * entity, struct media_pipeline * pipe)
Mark a pipeline as streaming

Parameters

struct media_entity * entity Starting entity

struct media_pipeline * pipe Media pipeline to be assigned to all entities in the pipeline.

Description

..note:: This is the non-locking version of `media_pipeline_start()`

void **media_pipeline_stop**(struct media_entity * entity)
Mark a pipeline as not streaming

Parameters

struct media_entity * entity Starting entity

Description

Mark all entities connected to a given entity through enabled links, either directly or indirectly, as not streaming. The media_entity pipe field is reset to `NULL`.

If multiple calls to `media_pipeline_start()` have been made, the same number of calls to this function are required to mark the pipeline as not streaming.

void **__media_pipeline_stop**(struct media_entity * entity)
Mark a pipeline as not streaming

Parameters

struct media_entity * entity Starting entity

Description

Note: This is the non-locking version of `media_pipeline_stop()`

---

struct media_intf_devnode * **media_devnode_create**(struct media_device * mdev, u32 type, u32 flags, u32 major, u32 minor)
creates and initializes a device node interface

Parameters

struct media_device * mdev pointer to struct `media_device`

u32 type type of the interface, as given by include/uapi/linux/media.h (seek for `MEDIA_INTF_T_*`) macros.

u32 flags Interface flags, as defined in include/uapi/linux/media.h (seek for `MEDIA_INTF_FL_*`)

u32 major Device node major number.

u32 minor Device node minor number.

Return

if succeeded, returns a pointer to the newly allocated `media_intf_devnode` pointer.

Description

Note: Currently, no flags for `media_interface` is defined.

---

void **media_devnode_remove**(struct media_intf_devnode * devnode)
removes a device node interface

Parameters

struct media_intf_devnode * devnode pointer to `media_intf_devnode` to be freed.

Description

When a device node interface is removed, all links to it are automatically removed.

media_create_intf_link(struct media_entity * entity, struct media_interface * intf, u32 flags)
creates a link between an entity and an interface

Parameters

struct media_entity * entity pointer to `media_entity`

struct media_interface * intf pointer to `media_interface`

u32 flags Link flags, as defined in include/uapi/linux/media.h (seek for MEDIA_LNK_FL_*)

Description

Valid values for flags:

MEDIA_LNK_FL_ENABLED Indicates that the interface is connected to the entity hardware. That's the default value for interfaces. An interface may be disabled if the hardware is busy due to the usage of some other interface that it is currently controlling the hardware.

A typical example is an hybrid TV device that handle only one type of stream on a given time. So, when the digital TV is streaming, the V4L2 interfaces won't be enabled, as such device is not able to also stream analog TV or radio.

Note: Before calling this function, media_devnode_create() should be called for the interface and media_device_register_entity() should be called for the interface that will be part of the link.

---

void **__media_remove_intf_link**(struct media_link * link)
remove a single interface link

Parameters

struct media_link * link pointer to media_link.

Description

Note: This is an unlocked version of media_remove_intf_link()

---

void **media_remove_intf_link**(struct media_link * link)
remove a single interface link

Parameters

struct media_link * link pointer to media_link.

Description

Note: Prefer to use this one, instead of __media_remove_intf_link()

---

void **__media_remove_intf_links**(struct media_interface * intf)
remove all links associated with an interface

Parameters

struct media_interface * intf pointer to media_interface

Description

Note: This is an unlocked version of media_remove_intf_links().

---

void **media_remove_intf_links**(struct media_interface * intf)
remove all links associated with an interface

Parameters

struct media_interface * intf pointer to `media_interface`

Description

Note:

1) This is called automatically when an entity is unregistered via `media_device_register_entity()` and by `media_devnode_remove()`.
2) Prefer to use this one, instead of `__media_remove_intf_links()`.

---

media_entity_call(entity, operation, args)
Calls a struct media_entity_operations operation on an entity

Parameters

entity entity where the **operation** will be called

operation type of the operation. Should be the name of a member of struct `media_entity_operations`.

args variable arguments

Description

This helper function will check if **operation** is not `NULL`. On such case, it will issue a call to **operation**(**entity**, **args**).

enum **media_request_state**
media request state

Constants

MEDIA_REQUEST_STATE_IDLE Idle

MEDIA_REQUEST_STATE_VALIDATING Validating the request, no state changes allowed

MEDIA_REQUEST_STATE_QUEUED Queued

MEDIA_REQUEST_STATE_COMPLETE Completed, the request is done

MEDIA_REQUEST_STATE_CLEANING Cleaning, the request is being re-inited

MEDIA_REQUEST_STATE_UPDATING The request is being updated, i.e. request objects are being added, modified or removed

NR_OF_MEDIA_REQUEST_STATE The number of media request states, used internally for sanity check purposes

struct **media_request**
Media device request

Definition

```
struct media_request {
  struct media_device *mdev;
  struct kref kref;
  char debug_str[TASK_COMM_LEN + 11];
  enum media_request_state state;
  unsigned int updating_count;
  unsigned int access_count;
  struct list_head objects;
  unsigned int num_incomplete_objects;
  wait_queue_head_t poll_wait;
  spinlock_t lock;
};
```

Members

mdev Media device this request belongs to

kref Reference count

debug_str Prefix for debug messages (process name:fd)

state The state of the request

updating_count count the number of request updates that are in progress

access_count count the number of request accesses that are in progress

objects List of **struct** media_request_object request objects

num_incomplete_objects The number of incomplete objects in the request

poll_wait Wait queue for poll

lock Serializes access to this struct

int **media_request_lock_for_access**(struct media_request * req)
Lock the request to access its objects

Parameters

struct media_request * req The media request

Description

Use before accessing a completed request. A reference to the request must be held during the access. This usually takes place automatically through a file handle. Use **media_request_unlock_for_access** when done.

void **media_request_unlock_for_access**(struct media_request * req)
Unlock a request previously locked for access

Parameters

struct media_request * req The media request

Description

Unlock a request that has previously been locked using **media_request_lock_for_access**.

int **media_request_lock_for_update**(struct media_request * req)
Lock the request for updating its objects

Parameters

struct media_request * req The media request

Description

Use before updating a request, i.e. adding, modifying or removing a request object in it. A reference to the request must be held during the update. This usually takes place automatically through a file handle. Use **media_request_unlock_for_update** when done.

void **media_request_unlock_for_update**(struct media_request * req)
Unlock a request previously locked for update

Parameters

struct media_request * req The media request

Description

Unlock a request that has previously been locked using **media_request_lock_for_update**.

void **media_request_get**(struct media_request * req)
Get the media request

Parameters

struct media_request * req The media request

Description

Get the media request.

void **media_request_put**(struct media_request * req)
Put the media request

Parameters

struct media_request * req The media request

Description

Put the media request. The media request will be released when the refcount reaches 0.

struct media_request * **media_request_get_by_fd**(struct media_device * mdev, int request_fd)
Get a media request by fd

Parameters

struct media_device * mdev Media device this request belongs to

int request_fd The file descriptor of the request

Description

Get the request represented by **request_fd** that is owned by the media device.

Return a -EBADR error pointer if requests are not supported by this driver. Return -EINVAL if the request was not found. Return the pointer to the request if found: the caller will have to call **media_request_put** when it finished using the request.

int **media_request_alloc**(struct media_device * mdev, int * alloc_fd)
Allocate the media request

Parameters

struct media_device * mdev Media device this request belongs to

int * alloc_fd Store the request's file descriptor in this int

Description

Allocated the media request and put the fd in **alloc_fd**.

struct **media_request_object_ops**
Media request object operations

Definition

```
struct media_request_object_ops {
  int (*prepare)(struct media_request_object *object);
  void (*unprepare)(struct media_request_object *object);
  void (*queue)(struct media_request_object *object);
  void (*unbind)(struct media_request_object *object);
  void (*release)(struct media_request_object *object);
};
```

Members

prepare Validate and prepare the request object, optional.

unprepare Unprepare the request object, optional.

queue Queue the request object, optional.

unbind Unbind the request object, optional.

release Release the request object, required.

struct **media_request_object**
An opaque object that belongs to a media request

Definition

```
struct media_request_object {
  const struct media_request_object_ops *ops;
  void *priv;
  struct media_request *req;
  struct list_head list;
  struct kref kref;
  bool completed;
};
```

Members

ops object's operations

priv object's priv pointer

req the request this object belongs to (can be NULL)

list List entry of the object for **struct** media_request

kref Reference count of the object, acquire before releasing req->lock

completed If true, then this object was completed.

Description

An object related to the request. This struct is always embedded in another struct that contains the actual data for this request object.

void **media_request_object_get**(struct media_request_object * obj)
Get a media request object

Parameters

struct media_request_object * obj The object

Description

Get a media request object.

void **media_request_object_put**(struct media_request_object * obj)
Put a media request object

Parameters

struct media_request_object * obj The object

Description

Put a media request object. Once all references are gone, the object's memory is released.

struct media_request_object * **media_request_object_find**(struct media_request * req, const struct media_request_object_ops * ops, void * priv)
Find an object in a request

Parameters

struct media_request * req The media request

const struct media_request_object_ops * ops Find an object with this ops value

void * priv Find an object with this priv value

Description

Both **ops** and **priv** must be non-NULL.

Returns the object pointer or NULL if not found. The caller must call `media_request_object_put()` once it finished using the object.

Since this function needs to walk the list of objects it takes the **req->lock** spin lock to make this safe.

void **media_request_object_init**(struct media_request_object * obj)
Initialise a media request object

Parameters

struct media_request_object * obj The object

Description

Initialise a media request object. The object will be released using the release callback of the ops once it has no references (this function initialises references to one).

int **media_request_object_bind**(struct media_request * req, const struct media_request_object_ops * ops, void * priv, bool is_buffer, struct media_request_object * obj)
Bind a media request object to a request

Parameters

struct media_request * req The media request

const struct media_request_object_ops * ops The object ops for this object

void * priv A driver-specific priv pointer associated with this object

bool is_buffer Set to true if the object a buffer object.

struct media_request_object * obj The object

Description

Bind this object to the request and set the ops and priv values of the object so it can be found later with `media_request_object_find()`.

Every bound object must be unbound or completed by the kernel at some point in time, otherwise the request will never complete. When the request is released all completed objects will be unbound by the request core code.

Buffer objects will be added to the end of the request's object list, non-buffer objects will be added to the front of the list. This ensures that all buffer objects are at the end of the list and that all non-buffer objects that they depend on are processed first.

void **media_request_object_unbind**(struct media_request_object * obj)
Unbind a media request object

Parameters

struct media_request_object * obj The object

Description

Unbind the media request object from the request.

void **media_request_object_complete**(struct media_request_object * obj)
Mark the media request object as complete

Parameters

struct media_request_object * obj The object

Description

Mark the media request object as complete. Only bound objects can be completed.

struct media_device * **media_device_usb_allocate**(struct usb_device * udev, const char * module_name, struct module * owner)

Allocate and return struct `media` device

Parameters

struct usb_device * udev struct `usb_device` pointer

const char * module_name should be filled with `KBUILD_MODNAME`

struct module * owner struct module pointer `THIS_MODULE` for the driver. `THIS_MODULE` is null for a built-in driver. It is safe even when `THIS_MODULE` is null.

Description

This interface should be called to allocate a Media Device when multiple drivers share usb_device and the media device. This interface allocates `media_device` structure and calls `media_device_usb_init()` to initialize it.

void **media_device_delete**(struct media_device * mdev, const char * module_name, struct module * owner)

Release media device. Calls kref_put().

Parameters

struct media_device * mdev struct `media_device` pointer

const char * module_name should be filled with `KBUILD_MODNAME`

struct module * owner struct module pointer `THIS_MODULE` for the driver. `THIS_MODULE` is null for a built-in driver. It is safe even when `THIS_MODULE` is null.

Description

This interface should be called to put Media Device Instance kref.

53.5 CEC Kernel Support

The CEC framework provides a unified kernel interface for use with HDMI CEC hardware. It is designed to handle a multiple types of hardware (receivers, transmitters, USB dongles). The framework also gives the option to decide what to do in the kernel driver and what should be handled by userspace applications. In addition it integrates the remote control passthrough feature into the kernel's remote control framework.

53.5.1 The CEC Protocol

The CEC protocol enables consumer electronic devices to communicate with each other through the HDMI connection. The protocol uses logical addresses in the communication. The logical address is strictly connected with the functionality provided by the device. The TV acting as the communication hub is always assigned address 0. The physical address is determined by the physical connection between devices.

The CEC framework described here is up to date with the CEC 2.0 specification. It is documented in the HDMI 1.4 specification with the new 2.0 bits documented in the HDMI 2.0 specification. But for most of the features the freely available HDMI 1.3a specification is sufficient:

http://www.microprocessor.org/HDMISpecification13a.pdf

53.5.2 CEC Adapter Interface

The struct cec_adapter represents the CEC adapter hardware. It is created by calling cec_allocate_adapter() and deleted by calling cec_delete_adapter():

```
struct cec_adapter *cec_allocate_adapter(const struct cec_adap_ops *ops, void *p
const char *name, u32 caps, u8 available_las);

void cec_delete_adapter(struct cec_adapter *adap);
```

To create an adapter you need to pass the following information:

ops: adapter operations which are called by the CEC framework and that you have to implement.

priv: will be stored in adap->priv and can be used by the adapter ops. Use cec_get_drvdata(adap) to get the priv pointer.

name: the name of the CEC adapter. Note: this name will be copied.

caps: capabilities of the CEC adapter. These capabilities determine the capabilities of the hardware and which parts are to be handled by userspace and which parts are handled by kernelspace. The capabilities are returned by CEC_ADAP_G_CAPS.

available_las: the number of simultaneous logical addresses that this adapter can handle. Must be 1 <= available_las <= CEC_MAX_LOG_ADDRS.

To obtain the priv pointer use this helper function:

```
void *cec_get_drvdata(const struct cec_adapter *adap);
```

To register the /dev/cecX device node and the remote control device (if CEC_CAP_RC is set) you call:

```
int cec_register_adapter(struct cec_adapter *adap, struct device *parent);
```

where parent is the parent device.

To unregister the devices call:

```
void cec_unregister_adapter(struct cec_adapter *adap);
```

Note: if cec_register_adapter() fails, then call cec_delete_adapter() to clean up. But if cec_register_adapter() succeeded, then only call cec_unregister_adapter() to clean up, never cec_delete_adapter(). The unregister function will delete the adapter automatically once the last user of that /dev/cecX device has closed its file handle.

53.5.3 Implementing the Low-Level CEC Adapter

The following low-level adapter operations have to be implemented in your driver:

struct **cec_adap_ops**

```
struct cec_adap_ops
{
        /* Low-level callbacks */
        int (*adap_enable)(struct cec_adapter *adap, bool enable);
        int (*adap_monitor_all_enable)(struct cec_adapter *adap, bool␣
↪enable);
        int (*adap_monitor_pin_enable)(struct cec_adapter *adap, bool␣
↪enable);
        int (*adap_log_addr)(struct cec_adapter *adap, u8 logical_addr);
        int (*adap_transmit)(struct cec_adapter *adap, u8 attempts,
                             u32 signal_free_time, struct cec_msg *msg);
        void (*adap_status)(struct cec_adapter *adap, struct seq_file␣
↪*file);
        void (*adap_free)(struct cec_adapter *adap);

        /* Error injection callbacks */
        ...

        /* High-level callbacks */
        ...
};
```

The seven low-level ops deal with various aspects of controlling the CEC adapter hardware:

To enable/disable the hardware:

int (*adap_enable)(struct cec_adapter *adap, bool enable);

This callback enables or disables the CEC hardware. Enabling the CEC hardware means powering it up in a state where no logical addresses are claimed. This op assumes that the physical address (adap->phys_addr) is valid when enable is true and will not change while the CEC adapter remains enabled. The initial state of the CEC adapter after calling cec_allocate_adapter() is disabled.

Note that adap_enable must return 0 if enable is false.

To enable/disable the ‘monitor all’ mode:

int (*adap_monitor_all_enable)(struct cec_adapter *adap, bool enable);

If enabled, then the adapter should be put in a mode to also monitor messages that not for us. Not all hardware supports this and this function is only called

if the CEC_CAP_MONITOR_ALL capability is set. This callback is optional (some hardware may always be in ‘monitor all’ mode).

Note that adap_monitor_all_enable must return 0 if enable is false.

To enable/disable the ‘monitor pin’ mode:

```
int (*adap_monitor_pin_enable)(struct cec_adapter *adap, bool enable);
```

If enabled, then the adapter should be put in a mode to also monitor CEC pin changes. Not all hardware supports this and this function is only called if the CEC_CAP_MONITOR_PIN capability is set. This callback is optional (some hardware may always be in ‘monitor pin’ mode).

Note that adap_monitor_pin_enable must return 0 if enable is false.

To program a new logical address:

```
int (*adap_log_addr)(struct cec_adapter *adap, u8 logical_addr);
```

If logical_addr == CEC_LOG_ADDR_INVALID then all programmed logical addresses are to be erased. Otherwise the given logical address should be programmed. If the maximum number of available logical addresses is exceeded, then it should return -ENXIO. Once a logical address is programmed the CEC hardware can receive directed messages to that address.

Note that adap_log_addr must return 0 if logical_addr is CEC_LOG_ADDR_INVALID.

To transmit a new message:

```
int (*adap_transmit)(struct cec_adapter *adap, u8 attempts,
u32 signal_free_time, struct cec_msg *msg);
```

This transmits a new message. The attempts argument is the suggested number of attempts for the transmit.

The signal_free_time is the number of data bit periods that the adapter should wait when the line is free before attempting to send a message. This value depends on whether this transmit is a retry, a message from a new initiator or a new message for the same initiator. Most hardware will handle this automatically, but in some cases this information is needed.

The CEC_FREE_TIME_TO_USEC macro can be used to convert signal_free_time to microseconds (one data bit period is 2.4 ms).

To log the current CEC hardware status:

```
void (*adap_status)(struct cec_adapter *adap, struct seq_file *file);
```

This optional callback can be used to show the status of the CEC hardware. The status is available through debugfs: cat /sys/kernel/debug/cec/cecX/status

To free any resources when the adapter is deleted:

```
void (*adap_free)(struct cec_adapter *adap);
```

This optional callback can be used to free any resources that might have been allocated by the driver. It’s called from cec_delete_adapter.

Your adapter driver will also have to react to events (typically interrupt driven) by calling into the framework in the following situations:

When a transmit finished (successfully or otherwise):

```
void cec_transmit_done(struct cec_adapter *adap, u8 status, u8 arb_lost_cnt,
u8 nack_cnt, u8 low_drive_cnt, u8 error_cnt);
```

or:

```
void cec_transmit_attempt_done(struct cec_adapter *adap, u8 status);
```

The status can be one of:

CEC_TX_STATUS_OK: the transmit was successful.

CEC_TX_STATUS_ARB_LOST: arbitration was lost: another CEC initiator took control of the CEC line and you lost the arbitration.

CEC_TX_STATUS_NACK: the message was nacked (for a directed message) or acked (for a broadcast message). A retransmission is needed.

CEC_TX_STATUS_LOW_DRIVE: low drive was detected on the CEC bus. This indicates that a follower detected an error on the bus and requested a retransmission.

CEC_TX_STATUS_ERROR: some unspecified error occurred: this can be one of ARB_LOST or LOW_DRIVE if the hardware cannot differentiate or something else entirely. Some hardware only supports OK and FAIL as the result of a transmit, i.e. there is no way to differentiate between the different possible errors. In that case map FAIL to CEC_TX_STATUS_NACK and not to CEC_TX_STATUS_ERROR.

CEC_TX_STATUS_MAX_RETRIES: could not transmit the message after trying multiple times. Should only be set by the driver if it has hardware support for retrying messages. If set, then the framework assumes that it doesn't have to make another attempt to transmit the message since the hardware did that already.

The hardware must be able to differentiate between OK, NACK and 'something else'.

The *_cnt arguments are the number of error conditions that were seen. This may be 0 if no information is available. Drivers that do not support hardware retry can just set the counter corresponding to the transmit error to 1, if the hardware does support retry then either set these counters to 0 if the hardware provides no feedback of which errors occurred and how many times, or fill in the correct values as reported by the hardware.

Be aware that calling these functions can immediately start a new transmit if there is one pending in the queue. So make sure that the hardware is in a state where new transmits can be started before calling these functions.

The cec_transmit_attempt_done() function is a helper for cases where the hardware never retries, so the transmit is always for just a single attempt. It will call cec_transmit_done() in turn, filling in 1 for the count argument corresponding to the status. Or all 0 if the status was OK.

When a CEC message was received:

```
void cec_received_msg(struct cec_adapter *adap, struct cec_msg *msg);
```

Speaks for itself.

53.5.4 Implementing the interrupt handler

Typically the CEC hardware provides interrupts that signal when a transmit finished and whether it was successful or not, and it provides and interrupt when a CEC message was received.

The CEC driver should always process the transmit interrupts first before handling the receive interrupt. The framework expects to see the cec_transmit_done call before the cec_received_msg call, otherwise it can get confused if the received message was in reply to the transmitted message.

53.5.5 Optional: Implementing Error Injection Support

If the CEC adapter supports Error Injection functionality, then that can be exposed through the Error Injection callbacks:

```
struct cec_adap_ops {
        /* Low-level callbacks */
        ...

        /* Error injection callbacks */
        int (*error_inj_show)(struct cec_adapter *adap, struct seq_file␣
↪*sf);
        bool (*error_inj_parse_line)(struct cec_adapter *adap, char *line);

        /* High-level CEC message callback */
        ...
};
```

If both callbacks are set, then an `error-inj` file will appear in debugfs. The basic syntax is as follows:

Leading spaces/tabs are ignored. If the next character is a # or the end of the line was reached, then the whole line is ignored. Otherwise a command is expected.

This basic parsing is done in the CEC Framework. It is up to the driver to decide what commands to implement. The only requirement is that the command `clear` without any arguments must be implemented and that it will remove all current error injection commands.

This ensures that you can always do `echo clear >error-inj` to clear any error injections without having to know the details of the driver-specific commands.

Note that the output of `error-inj` shall be valid as input to `error-inj`. So this must work:

```
$ cat error-inj >einj.txt
$ cat einj.txt >error-inj
```

The first callback is called when this file is read and it should show the the current error injection state:

`int (*error_inj_show)(struct cec_adapter *adap, struct seq_file *sf);`

It is recommended that it starts with a comment block with basic usage information. It returns 0 for success and an error otherwise.

The second callback will parse commands written to the `error-inj` file:

`bool (*error_inj_parse_line)(struct cec_adapter *adap, char *line);`

The `line` argument points to the start of the command. Any leading spaces or tabs have already been skipped. It is a single line only (so there are no embedded newlines) and it is 0-terminated. The callback is free to modify the contents of the buffer. It is only called for lines containing a command, so this callback is never called for empty lines or comment lines.

Return true if the command was valid or false if there were syntax errors.

53.5.6 Implementing the High-Level CEC Adapter

The low-level operations drive the hardware, the high-level operations are CEC protocol driven. The following high-level callbacks are available:

```
struct cec_adap_ops {
        /* Low-level callbacks */
        ...

        /* Error injection callbacks */
        ...

        /* High-level CEC message callback */
        int (*received)(struct cec_adapter *adap, struct cec_msg *msg);
};
```

The received() callback allows the driver to optionally handle a newly received CEC message

`int (*received)(struct cec_adapter *adap, struct cec_msg *msg);`

If the driver wants to process a CEC message, then it can implement this callback. If it doesn't want to handle this message, then it should return -ENOMSG, otherwise the CEC framework assumes it processed this message and it will not do anything with it.

53.5.7 CEC framework functions

CEC Adapter drivers can call the following CEC framework functions:

`int cec_transmit_msg(struct cec_adapter *adap, struct cec_msg *msg, bool block);`

Transmit a CEC message. If block is true, then wait until the message has been transmitted, otherwise just queue it and return.

`void cec_s_phys_addr(struct cec_adapter *adap, u16 phys_addr, bool block);`

Change the physical address. This function will set adap->phys_addr and send an event if it has changed. If cec_s_log_addrs() has been called and the physical address has become valid, then the CEC framework will start claiming the logical addresses. If block is true, then this function won't return until this process has finished.

When the physical address is set to a valid value the CEC adapter will be enabled (see the adap_enable op). When it is set to CEC_PHYS_ADDR_INVALID, then the CEC adapter will be disabled. If you change a valid physical address to another valid physical address, then this function will first set the address to CEC_PHYS_ADDR_INVALID before enabling the new physical address.

void cec_s_phys_addr_from_edid(struct cec_adapter *adap, const struct edid *edid);

A helper function that extracts the physical address from the edid struct and calls cec_s_phys_addr() with that address, or CEC_PHYS_ADDR_INVALID if the EDID did not contain a physical address or edid was a NULL pointer.

int cec_s_log_addrs(struct cec_adapter *adap, struct cec_log_addrs *log_addrs, bool block);

Claim the CEC logical addresses. Should never be called if CEC_CAP_LOG_ADDRS is set. If block is true, then wait until the logical addresses have been claimed, otherwise just queue it and return. To unconfigure all logical addresses call this function with log_addrs set to NULL or with log_addrs->num_log_addrs set to 0. The block argument is ignored when unconfiguring. This function will just return if the physical address is invalid. Once the physical address becomes valid, then the framework will attempt to claim these logical addresses.

53.5.8 CEC Pin framework

Most CEC hardware operates on full CEC messages where the software provides the message and the hardware handles the low-level CEC protocol. But some hardware only drives the CEC pin and software has to handle the low-level CEC protocol. The CEC pin framework was created to handle such devices.

Note that due to the close-to-realtime requirements it can never be guaranteed to work 100%. This framework uses highres timers internally, but if a timer goes off too late by more than 300 microseconds wrong results can occur. In reality it appears to be fairly reliable.

One advantage of this low-level implementation is that it can be used as a cheap CEC analyser, especially if interrupts can be used to detect CEC pin transitions from low to high or vice versa.

struct **cec_pin_ops**
low-level CEC pin operations

Definition

```
struct cec_pin_ops {
  int (*read)(struct cec_adapter *adap);
  void (*low)(struct cec_adapter *adap);
  void (*high)(struct cec_adapter *adap);
  bool (*enable_irq)(struct cec_adapter *adap);
  void (*disable_irq)(struct cec_adapter *adap);
  void (*free)(struct cec_adapter *adap);
  void (*status)(struct cec_adapter *adap, struct seq_file *file);
  int (*read_hpd)(struct cec_adapter *adap);
  int (*read_5v)(struct cec_adapter *adap);
```

(continues on next page)

(continued from previous page)

```
  int (*received)(struct cec_adapter *adap, struct cec_msg *msg);
};
```

Members

read read the CEC pin. Returns > 0 if high, 0 if low, or an error if negative.

low drive the CEC pin low.

high stop driving the CEC pin. The pull-up will drive the pin high, unless someone else is driving the pin low.

enable_irq optional, enable the interrupt to detect pin voltage changes.

disable_irq optional, disable the interrupt.

free optional. Free any allocated resources. Called when the adapter is deleted.

status optional, log status information.

read_hpd optional. Read the HPD pin. Returns > 0 if high, 0 if low or an error if negative.

read_5v optional. Read the 5V pin. Returns > 0 if high, 0 if low or an error if negative.

received optional. High-level CEC message callback. Allows the driver to process CEC messages.

Description

These operations (except for the **received** op) are used by the cec pin framework to manipulate the CEC pin.

void **cec_pin_changed**(struct cec_adapter * adap, bool value)
update pin state from interrupt

Parameters

struct cec_adapter * adap pointer to the cec adapter

bool value when true the pin is high, otherwise it is low

Description

If changes of the CEC voltage are detected via an interrupt, then cec_pin_changed is called from the interrupt with the new value.

struct cec_adapter * **cec_pin_allocate_adapter**(const struct cec_pin_ops * pin_ops, void * priv, const char * name, u32 caps)
allocate a pin-based cec adapter

Parameters

const struct cec_pin_ops * pin_ops low-level pin operations

void * priv will be stored in adap->priv and can be used by the adapter ops. Use cec_get_drvdata(adap) to get the priv pointer.

const char * name the name of the CEC adapter. Note: this name will be copied.

u32 caps capabilities of the CEC adapter. This will be ORed with CEC_CAP_MONITOR_ALL and CEC_CAP_MONITOR_PIN.

Description

Allocate a cec adapter using the cec pin framework.

Return

a pointer to the cec adapter or an error pointer

53.5.9 CEC Notifier framework

Most drm HDMI implementations have an integrated CEC implementation and no notifier support is needed. But some have independent CEC implementations that have their own driver. This could be an IP block for an SoC or a completely separate chip that deals with the CEC pin. For those cases a drm driver can install a notifier and use the notifier to inform the CEC driver about changes in the physical address.

struct cec_notifier * **cec_notifier_conn_register**(struct device * hdmi_dev, const char * port_name, const struct cec_connector_info * conn_info)
find or create a new cec_notifier for the given HDMI device and connector tuple.

Parameters

struct device * hdmi_dev HDMI device that sends the events.

const char * port_name the connector name from which the event occurs. May be NULL if there is always only one HDMI connector created by the HDMI device.

const struct cec_connector_info * conn_info the connector info from which the event occurs (may be NULL)

Description

If a notifier for device **dev** and connector **port_name** already exists, then increase the refcount and return that notifier.

If it doesn't exist, then allocate a new notifier struct and return a pointer to that new struct.

Return NULL if the memory could not be allocated.

void **cec_notifier_conn_unregister**(struct cec_notifier * n)
decrease refcount and delete when the refcount reaches 0.

Parameters

struct cec_notifier * n notifier. If NULL, then this function does nothing.

struct cec_notifier * **cec_notifier_cec_adap_register**(struct device * hdmi_dev, const char * port_name, struct cec_adapter * adap)
find or create a new cec_notifier for the given device.

Parameters

struct device * hdmi_dev HDMI device that sends the events.

const char * port_name the connector name from which the event occurs. May be NULL if there is always only one HDMI connector created by the HDMI device.

struct cec_adapter * adap the cec adapter that registered this notifier.

Description

If a notifier for device **dev** and connector **port_name** already exists, then increase the refcount and return that notifier.

If it doesn't exist, then allocate a new notifier struct and return a pointer to that new struct.

Return NULL if the memory could not be allocated.

void **cec_notifier_cec_adap_unregister**(struct cec_notifier * n, struct cec_adapter * adap)
decrease refcount and delete when the refcount reaches 0.

Parameters

struct cec_notifier * n notifier. If NULL, then this function does nothing.

struct cec_adapter * adap the cec adapter that registered this notifier.

void **cec_notifier_set_phys_addr**(struct cec_notifier * n, u16 pa)
set a new physical address.

Parameters

struct cec_notifier * n the CEC notifier

u16 pa the CEC physical address

Description

Set a new CEC physical address. Does nothing if **n** == NULL.

void **cec_notifier_set_phys_addr_from_edid**(struct cec_notifier * n, const struct edid * edid)
set parse the PA from the EDID.

Parameters

struct cec_notifier * n the CEC notifier

const struct edid * edid the struct edid pointer

Description

Parses the EDID to obtain the new CEC physical address and set it. Does nothing if **n** == NULL.

struct device * **cec_notifier_parse_hdmi_phandle**(struct device * dev)
find the hdmi device from “hdmi-phandle”

Parameters

struct device * dev the device with the “hdmi-phandle” device tree property

Description

Returns the device pointer referenced by the “hdmi-phandle” property. Note that the refcount of the returned device is not incremented. This device pointer is only used as a key value in the notifier list, but it is never accessed by the CEC driver.

void **cec_notifier_phys_addr_invalidate**(struct cec_notifier * n)
set the physical address to INVALID

Parameters

struct cec_notifier * n the CEC notifier

Description

This is a simple helper function to invalidate the physical address. Does nothing if **n** == NULL.

53.6 MIPI CSI-2

CSI-2 is a data bus intended for transferring images from cameras to the host SoC. It is defined by the MIPI alliance.

53.6.1 Media bus formats

See v4l2-mbus-pixelcode for details on which media bus formats should be used for CSI-2 interfaces.

53.6.2 Transmitter drivers

CSI-2 transmitter, such as a sensor or a TV tuner, drivers need to provide the CSI-2 receiver with information on the CSI-2 bus configuration. These include the V4L2_CID_LINK_FREQ and V4L2_CID_PIXEL_RATE controls and (`v4l2_subdev_video_ops->s_stream()` callback). These interface elements must be present on the sub-device represents the CSI-2 transmitter.

The V4L2_CID_LINK_FREQ control is used to tell the receiver driver the frequency (and not the symbol rate) of the link. The V4L2_CID_PIXEL_RATE is may be used by the receiver to obtain the pixel rate the transmitter uses. The `v4l2_subdev_video_ops->s_stream()` callback provides an ability to start and stop the stream.

The value of the V4L2_CID_PIXEL_RATE is calculated as follows:

```
pixel_rate = link_freq * 2 * nr_of_lanes / bits_per_sample
```

where

Table 1: variables in pixel rate calculation

variable or constant	description
link_freq	The value of the V4L2_CID_LINK_FREQ integer64 menu item.
nr_of_lanes	Number of data lanes used on the CSI-2 link. This can be obtained from the OF endpoint configuration.
2	Two bits are transferred per clock cycle per lane.
bits_per_sample	Number of bits per sample.

The transmitter drivers must, if possible, configure the CSI-2 transmitter to LP-11 mode whenever the transmitter is powered on but not active, and maintain LP-11 mode until stream on. Only at stream on should the transmitter activate the clock on the clock lane and transition to HS mode.

Some transmitters do this automatically but some have to be explicitly programmed to do so, and some are unable to do so altogether due to hardware constraints.

Stopping the transmitter

A transmitter stops sending the stream of images as a result of calling the `.s_stream()` callback. Some transmitters may stop the stream at a frame boundary whereas others stop immediately, effectively leaving the current frame unfinished. The receiver driver should not make assumptions either way, but function properly in both cases.

53.6.3 Receiver drivers

Before the receiver driver may enable the CSI-2 transmitter by using the `v4l2_subdev_video_ops->s_stream()`, it must have powered the transmitter up by using the `v4l2_subdev_core_ops->s_power()` callback. This may take place either indirectly by using `v4l2_pipeline_pm_get()` or directly.

53.6.4 Formats

The media bus pixel codes document parallel formats. Should the pixel data be transported over a serial bus, the media bus pixel code that describes a parallel format that transfers a sample on a single clock cycle is used.

53.7 Media driver-specific documentation

53.7.1 Video4Linux (V4L) drivers

The bttv driver

bttv and sound mini howto

There are a lot of different bt848/849/878/879 based boards available. Making video work often is not a big deal, because this is handled completely by the bt8xx chip, which is common on all boards. But sound is handled in slightly different ways on each board.

To handle the grabber boards correctly, there is a array tvcards[] in bttv-cards.c, which holds the information required for each board. Sound will work only, if the correct entry is used (for video it often makes no difference). The bttv driver prints a line to the kernel log, telling which card type is used. Like this one:

```
bttv0: model: BT848(Hauppauge old) [autodetected]
```

You should verify this is correct. If it isn't, you have to pass the correct board type as insmod argument, `insmod bttv card=2` for example. The file /admin-guide/media/bttv-cardlist has a list of valid arguments for card.

If your card isn't listed there, you might check the source code for new entries which are not listed yet. If there isn't one for your card, you can check if one of the existing entries does work for you (just trial and error…).

Some boards have an extra processor for sound to do stereo decoding and other nice features. The msp34xx chips are used by Hauppauge for example. If your board has one, you might have to load a helper module like `msp3400` to make sound work. If there isn't one for the chip used on your board: Bad luck. Start writing a new one. Well, you might want to check the video4linux mailing list archive first …

Of course you need a correctly installed soundcard unless you have the speakers connected directly to the grabber board. Hint: check the mixer settings too. ALSA for example has everything muted by default.

How sound works in detail

Still doesn't work? Looks like some driver hacking is required. Below is a do-it-yourself description for you.

The bt8xx chips have 32 general purpose pins, and registers to control these pins. One register is the output enable register (`BT848_GPIO_OUT_EN`), it says which pins are actively driven by the bt848 chip. Another one is the data register (`BT848_GPIO_DATA`), where you can get/set the status if these pins. They can be used for input and output.

Most grabber board vendors use these pins to control an external chip which does the sound routing. But every board is a little different. These pins are also used

by some companies to drive remote control receiver chips. Some boards use the i2c bus instead of the gpio pins to connect the mux chip.

As mentioned above, there is a array which holds the required information for each known board. You basically have to create a new line for your board. The important fields are these two:

```
struct tvcard
{
      [ ... ]
      u32 gpiomask;
      u32 audiomux[6]; /* Tuner, Radio, external, internal, mute, stereo */
};
```

gpiomask specifies which pins are used to control the audio mux chip. The corresponding bits in the output enable register (BT848_GPIO_OUT_EN) will be set as these pins must be driven by the bt848 chip.

The audiomux[] array holds the data values for the different inputs (i.e. which pins must be high/low for tuner/mute/⋯). This will be written to the data register (BT848_GPIO_DATA) to switch the audio mux.

What you have to do is figure out the correct values for gpiomask and the audiomux array. If you have Windows and the drivers four your card installed, you might to check out if you can read these registers values used by the windows driver. A tool to do this is available from http://btwincap.sourceforge.net/download.html.

You might also dig around in the *.ini files of the Windows applications. You can have a look at the board to see which of the gpio pins are connected at all and then start trial-and-error ⋯

Starting with release 0.7.41 bttv has a number of insmod options to make the gpio debugging easier:

bttv_gpio=0/1	enable/disable gpio debug messages
gpiomask=n	set the gpiomask value
audiomux=i,j,...	set the values of the audiomux array
audioall=a	set the values of the audiomux array (one value for all array elements, useful to check out which effect the particular value has).

The messages printed with bttv_gpio=1 look like this:

```
bttv0: gpio: en=00000027, out=00000024 in=00ffffd8 [audio: off]

en  =   output _en_able register (BT848_GPIO_OUT_EN)
out =   _out_put bits of the data register (BT848_GPIO_DATA),
        i.e. BT848_GPIO_DATA & BT848_GPIO_OUT_EN
in  =   _in_put bits of the data register,
        i.e. BT848_GPIO_DATA & ~BT848_GPIO_OUT_EN
```

The cpia2 driver

Authors: Peter Pregler <Peter_Pregler@email.com>, Scott J. Bertin <scottbertin@yahoo.com>, and Jarl Totland <Jarl.Totland@bdc.no> for the original cpia driver, which this one was modelled from.

Notes to developers

- This is a driver version stripped of the 2.4 back compatibility and old MJPEG ioctl API. See cpia2.sf.net for 2.4 support.

Programmer's overview of cpia2 driver

Cpia2 is the second generation video coprocessor from VLSI Vision Ltd (now a division of ST Microelectronics). There are two versions. The first is the STV0672, which is capable of up to 30 frames per second (fps) in frame sizes up to CIF, and 15 fps for VGA frames. The STV0676 is an improved version, which can handle up to 30 fps VGA. Both coprocessors can be attached to two CMOS sensors - the vvl6410 CIF sensor and the vvl6500 VGA sensor. These will be referred to as the 410 and the 500 sensors, or the CIF and VGA sensors.

The two chipsets operate almost identically. The core is an 8051 processor, running two different versions of firmware. The 672 runs the VP4 video processor code, the 676 runs VP5. There are a few differences in register mappings for the two chips. In these cases, the symbols defined in the header files are marked with VP4 or VP5 as part of the symbol name.

The cameras appear externally as three sets of registers. Setting register values is the only way to control the camera. Some settings are interdependant, such as the sequence required to power up the camera. I will try to make note of all of these cases.

The register sets are called blocks. Block 0 is the system block. This section is always powered on when the camera is plugged in. It contains registers that control housekeeping functions such as powering up the video processor. The video processor is the VP block. These registers control how the video from the sensor is processed. Examples are timing registers, user mode (vga, qvga), scaling, cropping, framerates, and so on. The last block is the video compressor (VC). The video stream sent from the camera is compressed as Motion JPEG (JPEGA). The VC controls all of the compression parameters. Looking at the file cpia2_registers.h, you can get a full view of these registers and the possible values for most of them.

One or more registers can be set or read by sending a usb control message to the camera. There are three modes for this. Block mode requests a number of contiguous registers. Random mode reads or writes random registers with a tuple structure containing address/value pairs. The repeat mode is only used by VP4 to load a firmware patch. It contains a starting address and a sequence of bytes to be written into a gpio port.

The cx2341x driver

Memory at cx2341x chips

This section describes the cx2341x memory map and documents some of the register space.

Note: the memory long words are little-endian (‘intel format’).

Warning: This information was figured out from searching through the memory and registers, this information may not be correct and is certainly not complete, and was not derived from anything more than searching through the memory space with commands like:

```
ivtvctl -O min=0x02000000,max=0x020000ff
```

So take this as is, I’m always searching for more stuff, it’s a large register space :-).

Memory Map

The cx2341x exposes its entire 64M memory space to the PCI host via the PCI BAR0 (Base Address Register 0). The addresses here are offsets relative to the address held in BAR0.

```
0x00000000-0x00ffffff Encoder memory space
0x00000000-0x0003ffff Encode.rom
???-???         MPEG buffer(s)
???-???         Raw video capture buffer(s)
???-???         Raw audio capture buffer(s)
???-???         Display buffers (6 or 9)

0x01000000-0x01ffffff Decoder memory space
0x01000000-0x0103ffff Decode.rom
???-???         MPEG buffers(s)
0x0114b000-0x0115afff Audio.rom (deprecated?)

0x02000000-0x0200ffff Register Space
```

Registers

The registers occupy the 64k space starting at the 0x02000000 offset from BAR0. All of these registers are 32 bits wide.

```
DMA Registers 0x000-0xff:

0x00 - Control:
        0=reset/cancel, 1=read, 2=write, 4=stop
```

(continues on next page)

(continued from previous page)

```
0x04 - DMA status:
        1=read busy, 2=write busy, 4=read error, 8=write error, 16=link␣
↪list error
0x08 - pci DMA pointer for read link list
0x0c - pci DMA pointer for write link list
0x10 - read/write DMA enable:
        1=read enable, 2=write enable
0x14 - always 0xffffffff, if set any lower instability occurs, 0x00 crashes
0x18 - ??
0x1c - always 0x20 or 32, smaller values slow down DMA transactions
0x20 - always value of 0x780a010a
0x24-0x3c - usually just random values???
0x40 - Interrupt status
0x44 - Write a bit here and shows up in Interrupt status 0x40
0x48 - Interrupt Mask
0x4C - always value of 0xfffdffff,
        if changed to 0xffffffff DMA write interrupts break.
0x50 - always 0xffffffff
0x54 - always 0xffffffff (0x4c, 0x50, 0x54 seem like interrupt masks, are
        3 processors on chip, Java ones, VPU, SPU, APU, maybe these are the
        interrupt masks???).
0x60-0x7C - random values
0x80 - first write linked list reg, for Encoder Memory addr
0x84 - first write linked list reg, for pci memory addr
0x88 - first write linked list reg, for length of buffer in memory addr
        (|0x80000000 or this for last link)
0x8c-0xdc - rest of write linked list reg, 8 sets of 3 total, DMA goes here
        from linked list addr in reg 0x0c, firmware must push through or
        something.
0xe0 - first (and only) read linked list reg, for pci memory addr
0xe4 - first (and only) read linked list reg, for Decoder memory addr
0xe8 - first (and only) read linked list reg, for length of buffer
0xec-0xff - Nothing seems to be in these registers, 0xec-f4 are 0x00000000.
```

Memory locations for Encoder Buffers 0x700-0x7ff:

These registers show offsets of memory locations pertaining to each buffer area used for encoding, have to shift them by <<1 first.

- 0x07F8: Encoder SDRAM refresh
- 0x07FC: Encoder SDRAM pre-charge

Memory locations for Decoder Buffers 0x800-0x8ff:

These registers show offsets of memory locations pertaining to each buffer area used for decoding, have to shift them by <<1 first.

- 0x08F8: Decoder SDRAM refresh
- 0x08FC: Decoder SDRAM pre-charge

Other memory locations:

- 0x2800: Video Display Module control
- 0x2D00: AO (audio output?) control
- 0x2D24: Bytes Flushed

- 0x7000: LSB I2C write clock bit (inverted)
- 0x7004: LSB I2C write data bit (inverted)
- 0x7008: LSB I2C read clock bit
- 0x700c: LSB I2C read data bit
- 0x9008: GPIO get input state
- 0x900c: GPIO set output state
- 0x9020: GPIO direction (Bit7 (GPIO 0..7) - 0:input, 1:output)
- 0x9050: SPU control
- 0x9054: Reset HW blocks
- 0x9058: VPU control
- 0xA018: Bit6: interrupt pending?
- 0xA064: APU command

Interrupt Status Register

The definition of the bits in the interrupt status register 0x0040, and the interrupt mask 0x0048. If a bit is cleared in the mask, then we want our ISR to execute.

- bit 31 Encoder Start Capture
- bit 30 Encoder EOS
- bit 29 Encoder VBI capture
- bit 28 Encoder Video Input Module reset event
- bit 27 Encoder DMA complete
- bit 24 Decoder audio mode change detection event (through event notification)
- bit 22 Decoder data request
- bit 20 Decoder DMA complete
- bit 19 Decoder VBI re-insertion
- bit 18 Decoder DMA err (linked-list bad)

Missing documentation

- Encoder API post(?)
- Decoder API post(?)
- Decoder VTRACE event

The cx2341x firmware upload

This document describes how to upload the cx2341x firmware to the card.

How to find

See the web pages of the various projects that uses this chip for information on how to obtain the firmware.

The firmware stored in a Windows driver can be detected as follows:

- Each firmware image is 256k bytes.
- The 1st 32-bit word of the Encoder image is 0x0000da7
- The 1st 32-bit word of the Decoder image is 0x00003a7
- The 2nd 32-bit word of both images is 0xaa55bb66

How to load

- Issue the FWapi command to stop the encoder if it is running. Wait for the command to complete.
- Issue the FWapi command to stop the decoder if it is running. Wait for the command to complete.
- Issue the I2C command to the digitizer to stop emitting VSYNC events.
- Issue the FWapi command to halt the encoder’s firmware.
- Sleep for 10ms.
- Issue the FWapi command to halt the decoder’s firmware.
- Sleep for 10ms.
- Write 0x00000000 to register 0x2800 to stop the Video Display Module.
- Write 0x00000005 to register 0x2D00 to stop the AO (audio output?).
- Write 0x00000000 to register 0xA064 to ping? the APU.
- Write 0xFFFFFFFE to register 0x9058 to stop the VPU.
- Write 0xFFFFFFFF to register 0x9054 to reset the HW blocks.
- Write 0x00000001 to register 0x9050 to stop the SPU.
- Sleep for 10ms.
- Write 0x0000001A to register 0x07FC to init the Encoder SDRAM’s pre-charge.
- Write 0x80000640 to register 0x07F8 to init the Encoder SDRAM’s refresh to 1us.
- Write 0x0000001A to register 0x08FC to init the Decoder SDRAM’s pre-charge.

- Write 0x80000640 to register 0x08F8 to init the Decoder SDRAM' s refresh to 1us.
- Sleep for 512ms. (600ms is recommended)
- Transfer the encoder' s firmware image to offset 0 in Encoder memory space.
- Transfer the decoder' s firmware image to offset 0 in Decoder memory space.
- Use a read-modify-write operation to Clear bit 0 of register 0x9050 to re-enable the SPU.
- Sleep for 1 second.
- Use a read-modify-write operation to Clear bits 3 and 0 of register 0x9058 to re-enable the VPU.
- Sleep for 1 second.
- Issue status API commands to both firmware images to verify.

How to call the firmware API

The preferred calling convention is known as the firmware mailbox. The mailboxes are basically a fixed length array that serves as the call-stack.

Firmware mailboxes can be located by searching the encoder and decoder memory for a 16 byte signature. That signature will be located on a 256-byte boundary.

Signature:

```
0x78, 0x56, 0x34, 0x12, 0x12, 0x78, 0x56, 0x34,
0x34, 0x12, 0x78, 0x56, 0x56, 0x34, 0x12, 0x78
```

The firmware implements 20 mailboxes of 20 32-bit words. The first 10 are reserved for API calls. The second 10 are used by the firmware for event notification.

Index	Name
0	Flags
1	Command
2	Return value
3	Timeout
4-19	Parameter/Result

The flags are defined in the following table. The direction is from the perspective of the firmware.

Bit	Direction	Purpose
2	O	Firmware has processed the command.
1	I	Driver has finished setting the parameters.
0	I	Driver is using this mailbox.

The command is a 32-bit enumerator. The API specifics may be found in this chapter.

The return value is a 32-bit enumerator. Only two values are currently defined:

- 0=success
- -1=command undefined.

There are 16 parameters/results 32-bit fields. The driver populates these fields with values for all the parameters required by the call. The driver overwrites these fields with result values returned by the call.

The timeout value protects the card from a hung driver thread. If the driver doesn't handle the completed call within the timeout specified, the firmware will reset that mailbox.

To make an API call, the driver iterates over each mailbox looking for the first one available (bit 0 has been cleared). The driver sets that bit, fills in the command enumerator, the timeout value and any required parameters. The driver then sets the parameter ready bit (bit 1). The firmware scans the mailboxes for pending commands, processes them, sets the result code, populates the result value array with that call's return values and sets the call complete bit (bit 2). Once bit 2 is set, the driver should retrieve the results and clear all the flags. If the driver does not perform this task within the time set in the timeout register, the firmware will reset that mailbox.

Event notifications are sent from the firmware to the host. The host tells the firmware which events it is interested in via an API call. That call tells the firmware which notification mailbox to use. The firmware signals the host via an interrupt. Only the 16 Results fields are used, the Flags, Command, Return value and Timeout words are not used.

OSD firmware API description

Note: this API is part of the decoder firmware, so it's cx23415 only.

CX2341X_OSD_GET_FRAMEBUFFER

Enum: 65/0x41

Description

Return base and length of contiguous OSD memory.

Result[0]

OSD base address

Result[1]

OSD length

CX2341X_OSD_GET_PIXEL_FORMAT

Enum: 66/0x42

Description

Query OSD format

Result[0]

0=8bit index 1=16bit RGB 5:6:5 2=16bit ARGB 1:5:5:5 3=16bit ARGB 1:4:4:4 4=32bit ARGB 8:8:8:8

CX2341X_OSD_SET_PIXEL_FORMAT

Enum: 67/0x43

Description

Assign pixel format

Param[0]

- 0=8bit index
- 1=16bit RGB 5:6:5
- 2=16bit ARGB 1:5:5:5
- 3=16bit ARGB 1:4:4:4
- 4=32bit ARGB 8:8:8:8

CX2341X_OSD_GET_STATE

Enum: 68/0x44

Description

Query OSD state

Result[0]

- Bit 0 0=off, 1=on
- Bits 1:2 alpha control
- Bits 3:5 pixel format

CX2341X_OSD_SET_STATE

Enum: 69/0x45

Description

OSD switch

Param[0]

0=off, 1=on

CX2341X_OSD_GET_OSD_COORDS

Enum: 70/0x46

Description

Retrieve coordinates of OSD area blended with video

Result[0]

OSD buffer address

Result[1]

Stride in pixels

Result[2]

Lines in OSD buffer

Result[3]

Horizontal offset in buffer

Result[4]

Vertical offset in buffer

CX2341X_OSD_SET_OSD_COORDS

Enum: 71/0x47

Description

Assign the coordinates of the OSD area to blend with video

Param[0]

buffer address

Param[1]

buffer stride in pixels

Param[2]

lines in buffer

Param[3]

horizontal offset

Param[4]

vertical offset

CX2341X_OSD_GET_SCREEN_COORDS

Enum: 72/0x48

Description

Retrieve OSD screen area coordinates

Result[0]

top left horizontal offset

Result[1]

top left vertical offset

Result[2]

bottom right horizontal offset

Result[3]

bottom right vertical offset

CX2341X_OSD_SET_SCREEN_COORDS

Enum: 73/0x49

Description

Assign the coordinates of the screen area to blend with video

Param[0]

top left horizontal offset

Param[1]

top left vertical offset

Param[2]

bottom left horizontal offset

Param[3]

bottom left vertical offset

CX2341X_OSD_GET_GLOBAL_ALPHA

Enum: 74/0x4A

Description

Retrieve OSD global alpha

Result[0]

global alpha: 0=off, 1=on

Result[1]

bits 0:7 global alpha

CX2341X_OSD_SET_GLOBAL_ALPHA

Enum: 75/0x4B

Description

Update global alpha

Param[0]

global alpha: 0=off, 1=on

Param[1]

global alpha (8 bits)

Param[2]

local alpha: 0=on, 1=off

CX2341X_OSD_SET_BLEND_COORDS

Enum: 78/0x4C

Description

Move start of blending area within display buffer

Param[0]

horizontal offset in buffer

Param[1]

vertical offset in buffer

CX2341X_OSD_GET_FLICKER_STATE

Enum: 79/0x4F

Description

Retrieve flicker reduction module state

Result[0]

flicker state: 0=off, 1=on

CX2341X_OSD_SET_FLICKER_STATE

Enum: 80/0x50

Description

Set flicker reduction module state

Param[0]

State: 0=off, 1=on

CX2341X_OSD_BLT_COPY

Enum: 82/0x52

Description

BLT copy

Param[0]

```
'0000'  zero
'0001' ~destination AND ~source
'0010' ~destination AND  source
'0011' ~destination
'0100'  destination AND ~source
'0101'                  ~source
'0110'  destination XOR  source
'0111' ~destination OR  ~source
'1000' ~destination AND ~source
'1001'  destination XNOR source
'1010'                   source
'1011' ~destination OR   source
'1100'  destination
'1101'  destination OR  ~source
'1110'  destination OR   source
'1111'  one
```

Param[1]

Resulting alpha blending

- ‘01’ source_alpha
- ‘10’ destination_alpha
- ‘11’ source_alpha*destination_alpha+1 (zero if both source and destination alpha are zero)

Param[2]

```
'00' output_pixel = source_pixel

'01' if source_alpha=0:
         output_pixel = destination_pixel
     if 256 > source_alpha > 1:
         output_pixel = ((source_alpha + 1)*source_pixel +
                         (255 - source_alpha)*destination_pixel)/256

'10' if destination_alpha=0:
         output_pixel = source_pixel
      if 255 > destination_alpha > 0:
         output_pixel = ((255 - destination_alpha)*source_pixel +
                         (destination_alpha + 1)*destination_pixel)/256

'11' if source_alpha=0:
         source_temp = 0
     if source_alpha=255:
         source_temp = source_pixel*256
     if 255 > source_alpha > 0:
         source_temp = source_pixel*(source_alpha + 1)
     if destination_alpha=0:
```

(continues on next page)

(continued from previous page)

```
        destination_temp = 0
    if destination_alpha=255:
        destination_temp = destination_pixel*256
    if 255 > destination_alpha > 0:
        destination_temp = destination_pixel*(destination_alpha + 1)
    output_pixel = (source_temp + destination_temp)/256
```

Param[3]

width

Param[4]

height

Param[5]

destination pixel mask

Param[6]

destination rectangle start address

Param[7]

destination stride in dwords

Param[8]

source stride in dwords

Param[9]

source rectangle start address

CX2341X_OSD_BLT_FILL

Enum: 83/0x53

Description

BLT fill color

Param[0]

Same as Param[0] on API 0x52

Param[1]

Same as Param[1] on API 0x52

Param[2]

Same as Param[2] on API 0x52

Param[3]

width

Param[4]

height

Param[5]

destination pixel mask

Param[6]

destination rectangle start address

Param[7]

destination stride in dwords

Param[8]

color fill value

CX2341X_OSD_BLT_TEXT

Enum: 84/0x54

Description

BLT for 8 bit alpha text source

Param[0]

Same as Param[0] on API 0x52

Param[1]

Same as Param[1] on API 0x52

Param[2]

Same as Param[2] on API 0x52

Param[3]

width

Param[4]

height

Param[5]

destination pixel mask

Param[6]

destination rectangle start address

Param[7]

destination stride in dwords

Param[8]

source stride in dwords

Param[9]

source rectangle start address

Param[10]

color fill value

CX2341X_OSD_SET_FRAMEBUFFER_WINDOW

Enum: 86/0x56

Description

Positions the main output window on the screen. The coordinates must be such that the entire window fits on the screen.

Param[0]

window width

Param[1]

window height

Param[2]

top left window corner horizontal offset

Param[3]

top left window corner vertical offset

CX2341X_OSD_SET_CHROMA_KEY

Enum: 96/0x60

Description

Chroma key switch and color

Param[0]

state: 0=off, 1=on

Param[1]

color

CX2341X_OSD_GET_ALPHA_CONTENT_INDEX

Enum: 97/0x61

Description

Retrieve alpha content index

Result[0]

alpha content index, Range 0:15

CX2341X_OSD_SET_ALPHA_CONTENT_INDEX

Enum: 98/0x62

Description

Assign alpha content index

Param[0]

alpha content index, range 0:15

Encoder firmware API description

CX2341X_ENC_PING_FW

Enum: 128/0x80

Description

Does nothing. Can be used to check if the firmware is responding.

CX2341X_ENC_START_CAPTURE

Enum: 129/0x81

Description

Commences the capture of video, audio and/or VBI data. All encoding parameters must be initialized prior to this API call. Captures frames continuously or until a predefined number of frames have been captured.

Param[0]

Capture stream type:

- 0=MPEG
- 1=Raw
- 2=Raw passthrough
- 3=VBI

Param[1]

Bitmask:

- Bit 0 when set, captures YUV
- Bit 1 when set, captures PCM audio
- Bit 2 when set, captures VBI (same as param[0]=3)
- Bit 3 when set, the capture destination is the decoder (same as param[0]=2)
- Bit 4 when set, the capture destination is the host

Note: this parameter is only meaningful for RAW capture type.

CX2341X_ENC_STOP_CAPTURE

Enum: 130/0x82

Description

Ends a capture in progress

Param[0]

- 0=stop at end of GOP (generates IRQ)
- 1=stop immediate (no IRQ)

Param[1]

Stream type to stop, see param[0] of API 0x81

Param[2]

Subtype, see param[1] of API 0x81

CX2341X_ENC_SET_AUDIO_ID

Enum: 137/0x89

Description

Assigns the transport stream ID of the encoded audio stream

Param[0]

Audio Stream ID

CX2341X_ENC_SET_VIDEO_ID

Enum: 139/0x8B

Description

Set video transport stream ID

Param[0]

Video stream ID

CX2341X_ENC_SET_PCR_ID

Enum: 141/0x8D

Description

Assigns the transport stream ID for PCR packets

Param[0]

PCR Stream ID

CX2341X_ENC_SET_FRAME_RATE

Enum: 143/0x8F

Description

Set video frames per second. Change occurs at start of new GOP.

Param[0]

- 0=30fps
- 1=25fps

CX2341X_ENC_SET_FRAME_SIZE

Enum: 145/0x91

Description

Select video stream encoding resolution.

Param[0]

Height in lines. Default 480

Param[1]

Width in pixels. Default 720

CX2341X_ENC_SET_BIT_RATE

Enum: 149/0x95

Description

Assign average video stream bitrate.

Param[0]

0=variable bitrate, 1=constant bitrate

Param[1]

bitrate in bits per second

Param[2]

peak bitrate in bits per second, divided by 400

Param[3]

Mux bitrate in bits per second, divided by 400. May be 0 (default).

Param[4]

Rate Control VBR Padding

Param[5]

VBV Buffer used by encoder

Note:

1) Param[3] and Param[4] seem to be always 0
2) Param[5] doesn’t seem to be used.

CX2341X_ENC_SET_GOP_PROPERTIES

Enum: 151/0x97

Description

Setup the GOP structure

Param[0]

GOP size (maximum is 34)

Param[1]

Number of B frames between the I and P frame, plus 1. For example: IBBPBBPBBPBB –> GOP size: 12, number of B frames: 2+1 = 3

Note: GOP size must be a multiple of (B-frames + 1).

CX2341X_ENC_SET_ASPECT_RATIO

Enum: 153/0x99

Description

Sets the encoding aspect ratio. Changes in the aspect ratio take effect at the start of the next GOP.

Param[0]

- ‘0000’ forbidden
- ‘0001’ 1:1 square
- ‘0010’ 4:3
- ‘0011’ 16:9
- ‘0100’ 2.21:1
- ‘0101’ to ‘1111’ reserved

CX2341X_ENC_SET_DNR_FILTER_MODE

Enum: 155/0x9B

Description

Assign Dynamic Noise Reduction operating mode

Param[0]

Bit0: Spatial filter, set=auto, clear=manual Bit1: Temporal filter, set=auto, clear=manual

Param[1]

Median filter:

- 0=Disabled
- 1=Horizontal
- 2=Vertical
- 3=Horiz/Vert
- 4=Diagonal

CX2341X_ENC_SET_DNR_FILTER_PROPS

Enum: 157/0x9D

Description

These Dynamic Noise Reduction filter values are only meaningful when the respective filter is set to “manual” (See API 0x9B)

Param[0]

Spatial filter: default 0, range 0:15

Param[1]

Temporal filter: default 0, range 0:31

CX2341X_ENC_SET_CORING_LEVELS

Enum: 159/0x9F

Description

Assign Dynamic Noise Reduction median filter properties.

Param[0]

Threshold above which the luminance median filter is enabled. Default: 0, range 0:255

Param[1]

Threshold below which the luminance median filter is enabled. Default: 255, range 0:255

Param[2]

Threshold above which the chrominance median filter is enabled. Default: 0, range 0:255

Param[3]

Threshold below which the chrominance median filter is enabled. Default: 255, range 0:255

CX2341X_ENC_SET_SPATIAL_FILTER_TYPE

Enum: 161/0xA1

Description

Assign spatial prefilter parameters

Param[0]

Luminance filter

- 0=Off
- 1=1D Horizontal
- 2=1D Vertical
- 3=2D H/V Separable (default)
- 4=2D Symmetric non-separable

Param[1]

Chrominance filter

- 0=Off
- 1=1D Horizontal (default)

CX2341X_ENC_SET_VBI_LINE

Enum: 183/0xB7

Description

Selects VBI line number.

Param[0]

- Bits 0:4 line number
- Bit 31 0=top_field, 1=bottom_field
- Bits 0:31 all set specifies “all lines”

Param[1]

VBI line information features: 0=disabled, 1=enabled

Param[2]

Slicing: 0=None, 1=Closed Caption Almost certainly not implemented. Set to 0.

Param[3]

Luminance samples in this line. Almost certainly not implemented. Set to 0.

Param[4]

Chrominance samples in this line Almost certainly not implemented. Set to 0.

CX2341X_ENC_SET_STREAM_TYPE

Enum: 185/0xB9

Description

Assign stream type

Note: Transport stream is not working in recent firmwares. And in older firmwares the timestamps in the TS seem to be unreliable.

Param[0]

- 0=Program stream
- 1=Transport stream
- 2=MPEG1 stream
- 3=PES A/V stream
- 5=PES Video stream
- 7=PES Audio stream
- 10=DVD stream
- 11=VCD stream
- 12=SVCD stream
- 13=DVD_S1 stream

- 14=DVD_S2 stream

CX2341X_ENC_SET_OUTPUT_PORT

Enum: 187/0xBB

Description

Assign stream output port. Normally 0 when the data is copied through the PCI bus (DMA), and 1 when the data is streamed to another chip (pvrusb and cx88-blackbird).

Param[0]

- 0=Memory (default)
- 1=Streaming
- 2=Serial

Param[1]

Unknown, but leaving this to 0 seems to work best. Indications are that this might have to do with USB support, although passing anything but 0 only breaks things.

CX2341X_ENC_SET_AUDIO_PROPERTIES

Enum: 189/0xBD

Description

Set audio stream properties, may be called while encoding is in progress.

Note: All bitfields are consistent with ISO11172 documentation except bits 2:3 which ISO docs define as:

- '11' Layer I
- '10' Layer II
- '01' Layer III
- '00' Undefined

This discrepancy may indicate a possible error in the documentation. Testing indicated that only Layer II is actually working, and that the minimum bitrate should be 192 kbps.

Param[0]

Bitmask:

```
 0:1  '00' 44.1Khz
      '01' 48Khz
      '10' 32Khz
      '11' reserved

 2:3  '01'=Layer I
      '10'=Layer II

 4:7  Bitrate:
           Index | Layer I     | Layer II
           ------+-------------+------------
          '0000' | free format | free format
          '0001' |  32 kbit/s  |  32 kbit/s
          '0010' |  64 kbit/s  |  48 kbit/s
          '0011' |  96 kbit/s  |  56 kbit/s
          '0100' | 128 kbit/s  |  64 kbit/s
          '0101' | 160 kbit/s  |  80 kbit/s
          '0110' | 192 kbit/s  |  96 kbit/s
          '0111' | 224 kbit/s  | 112 kbit/s
          '1000' | 256 kbit/s  | 128 kbit/s
          '1001' | 288 kbit/s  | 160 kbit/s
          '1010' | 320 kbit/s  | 192 kbit/s
          '1011' | 352 kbit/s  | 224 kbit/s
          '1100' | 384 kbit/s  | 256 kbit/s
          '1101' | 416 kbit/s  | 320 kbit/s
          '1110' | 448 kbit/s  | 384 kbit/s

      .. note::

              For Layer II, not all combinations of total bitrate
              and mode are allowed. See ISO11172-3 3-Annex B,
              Table 3-B.2

 8:9  '00'=Stereo
      '01'=JointStereo
      '10'=Dual
      '11'=Mono

      .. note::

              The cx23415 cannot decode Joint Stereo properly.

10:11 Mode Extension used in joint_stereo mode.
      In Layer I and II they indicate which subbands are in
      intensity_stereo. All other subbands are coded in stereo.
          '00' subbands 4-31 in intensity_stereo, bound==4
          '01' subbands 8-31 in intensity_stereo, bound==8
          '10' subbands 12-31 in intensity_stereo, bound==12
          '11' subbands 16-31 in intensity_stereo, bound==16

12:13 Emphasis:
          '00' None
          '01' 50/15uS
```

(continues on next page)

(continued from previous page)

```
          '10' reserved
          '11' CCITT J.17

14    CRC:
          '0' off
          '1' on

15    Copyright:
          '0' off
          '1' on

16    Generation:
          '0' copy
          '1' original
```

CX2341X_ENC_HALT_FW

Enum: 195/0xC3

Description

The firmware is halted and no further API calls are serviced until the firmware is uploaded again.

CX2341X_ENC_GET_VERSION

Enum: 196/0xC4

Description

Returns the version of the encoder firmware.

Result[0]

Version bitmask: - Bits 0:15 build - Bits 16:23 minor - Bits 24:31 major

CX2341X_ENC_SET_GOP_CLOSURE

Enum: 197/0xC5

Description

Assigns the GOP open/close property.

Param[0]

- 0=Open
- 1=Closed

CX2341X_ENC_GET_SEQ_END

Enum: 198/0xC6

Description

Obtains the sequence end code of the encoder’ s buffer. When a capture is started a number of interrupts are still generated, the last of which will have Result[0] set to 1 and Result[1] will contain the size of the buffer.

Result[0]

State of the transfer (1 if last buffer)

Result[1]

If Result[0] is 1, this contains the size of the last buffer, undefined otherwise.

CX2341X_ENC_SET_PGM_INDEX_INFO

Enum: 199/0xC7

Description

Sets the Program Index Information. The information is stored as follows:

```
struct info {
        u32 length;             // Length of this frame
        u32 offset_low;         // Offset in the file of the
        u32 offset_high;        // start of this frame
        u32 mask1;              // Bits 0-2 are the type mask:
                                // 1=I, 2=P, 4=B
                                // 0=End of Program Index, other fields
                                //   are invalid.
        u32 pts;                // The PTS of the frame
```

(continues on next page)

(continued from previous page)

```
        u32 mask2;              // Bit 0 is bit 32 of the pts.
};
u32 table_ptr;
struct info index[400];
```

The table_ptr is the encoder memory address in the table were new entries will be written.

Note: This is a ringbuffer, so the table_ptr will wraparound.

Param[0]

Picture Mask: - 0=No index capture - 1=I frames - 3=I,P frames - 7=I,P,B frames

(Seems to be ignored, it always indexes I, P and B frames)

Param[1]

Elements requested (up to 400)

Result[0]

Offset in the encoder memory of the start of the table.

Result[1]

Number of allocated elements up to a maximum of Param[1]

CX2341X_ENC_SET_VBI_CONFIG

Enum: 200/0xC8

Description

Configure VBI settings

Param[0]

Bitmap:

```
0    Mode '0' Sliced, '1' Raw
1:3  Insertion:
         '000' insert in extension & user data
         '001' insert in private packets
         '010' separate stream and user data
         '111' separate stream and private data
8:15 Stream ID (normally 0xBD)
```

Param[1]

Frames per interrupt (max 8). Only valid in raw mode.

Param[2]

Total raw VBI frames. Only valid in raw mode.

Param[3]

Start codes

Param[4]

Stop codes

Param[5]

Lines per frame

Param[6]

Byte per line

Result[0]

Observed frames per interrupt in raw mode only. Rage 1 to Param[1]

Result[1]

Observed number of frames in raw mode. Range 1 to Param[2]

Result[2]

Memory offset to start or raw VBI data

CX2341X_ENC_SET_DMA_BLOCK_SIZE

Enum: 201/0xC9

Description

Set DMA transfer block size

Param[0]

DMA transfer block size in bytes or frames. When unit is bytes, supported block sizes are 2^7, 2^8 and 2^9 bytes.

Param[1]

Unit: 0=bytes, 1=frames

CX2341X_ENC_GET_PREV_DMA_INFO_MB_10

Enum: 202/0xCA

Description

Returns information on the previous DMA transfer in conjunction with bit 27 of the interrupt mask. Uses mailbox 10.

Result[0]

Type of stream

Result[1]

Address Offset

Result[2]

Maximum size of transfer

CX2341X_ENC_GET_PREV_DMA_INFO_MB_9

Enum: 203/0xCB

Description

Returns information on the previous DMA transfer in conjunction with bit 27 or 18 of the interrupt mask. Uses mailbox 9.

Result[0]

Status bits: - 0 read completed - 1 write completed - 2 DMA read error - 3 DMA write error - 4 Scatter-Gather array error

Result[1]

DMA type

Result[2]

Presentation Time Stamp bits 0..31

Result[3]

Presentation Time Stamp bit 32

CX2341X_ENC_SCHED_DMA_TO_HOST

Enum: 204/0xCC

Description

Setup DMA to host operation

Param[0]

Memory address of link list

Param[1]

Length of link list (wtf: what units ???)

Param[2]

DMA type (0=MPEG)

CX2341X_ENC_INITIALIZE_INPUT

Enum: 205/0xCD

Description

Initializes the video input

CX2341X_ENC_SET_FRAME_DROP_RATE

Enum: 208/0xD0

Description

For each frame captured, skip specified number of frames.

Param[0]

Number of frames to skip

CX2341X_ENC_PAUSE_ENCODER

Enum: 210/0xD2

Description

During a pause condition, all frames are dropped instead of being encoded.

Param[0]

- 0=Pause encoding
- 1=Continue encoding

CX2341X_ENC_REFRESH_INPUT

Enum: 211/0xD3

Description

Refreshes the video input

CX2341X_ENC_SET_COPYRIGHT

Enum: 212/0xD4

Description

Sets stream copyright property

Param[0]

- 0=Stream is not copyrighted
- 1=Stream is copyrighted

CX2341X_ENC_SET_EVENT_NOTIFICATION

Enum: 213/0xD5

Description

Setup firmware to notify the host about a particular event. Host must unmask the interrupt bit.

Param[0]

Event (0=refresh encoder input)

Param[1]

Notification 0=disabled 1=enabled

Param[2]

Interrupt bit

Param[3]

Mailbox slot, -1 if no mailbox required.

CX2341X_ENC_SET_NUM_VSYNC_LINES

Enum: 214/0xD6

Description

Depending on the analog video decoder used, this assigns the number of lines for field 1 and 2.

Param[0]

Field 1 number of lines: - 0x00EF for SAA7114 - 0x00F0 for SAA7115 - 0x0105 for Micronas

Param[1]

Field 2 number of lines: - 0x00EF for SAA7114 - 0x00F0 for SAA7115 - 0x0106 for Micronas

CX2341X_ENC_SET_PLACEHOLDER

Enum: 215/0xD7

Description

Provides a mechanism of inserting custom user data in the MPEG stream.

Param[0]

- 0=extension & user data
- 1=private packet with stream ID 0xBD

Param[1]

Rate at which to insert data, in units of frames (for private packet) or GOPs (for ext. & user data)

Param[2]

Number of data DWORDs (below) to insert

Param[3]

Custom data 0

Param[4]

Custom data 1

Param[5]

Custom data 2

Param[6]

Custom data 3

Param[7]

Custom data 4

Param[8]

Custom data 5

Param[9]

Custom data 6

Param[10]

Custom data 7

Param[11]

Custom data 8

CX2341X_ENC_MUTE_VIDEO

Enum: 217/0xD9

Description

Video muting

Param[0]

Bit usage:

```
 0      '0'=video not muted
        '1'=video muted, creates frames with the YUV color defined below
 1:7    Unused
 8:15   V chrominance information
16:23   U chrominance information
24:31   Y luminance information
```

CX2341X_ENC_MUTE_AUDIO

Enum: 218/0xDA

Description

Audio muting

Param[0]

- 0=audio not muted
- 1=audio muted (produces silent mpeg audio stream)

CX2341X_ENC_SET_VERT_CROP_LINE

Enum: 219/0xDB

Description

Something to do with ‘Vertical Crop Line’

Param[0]

If saa7114 and raw VBI capture and 60 Hz, then set to 10001. Else 0.

CX2341X_ENC_MISC

Enum: 220/0xDC

Description

Miscellaneous actions. Not known for 100% what it does. It's really a sort of ioctl call. The first parameter is a command number, the second the value.

Param[0]

Command number:

```
 1=set initial SCR value when starting encoding (works).
 2=set quality mode (apparently some test setting).
 3=setup advanced VIM protection handling.
   Always 1 for the cx23416 and 0 for cx23415.
 4=generate DVD compatible PTS timestamps
 5=USB flush mode
 6=something to do with the quantization matrix
 7=set navigation pack insertion for DVD: adds 0xbf (private stream 2)
   packets to the MPEG. The size of these packets is 2048 bytes (including
   the header of 6 bytes: 0x000001bf + length). The payload is zeroed and
   it is up to the application to fill them in. These packets are␣
↪apparently
   inserted every four frames.
 8=enable scene change detection (seems to be a failure)
 9=set history parameters of the video input module
10=set input field order of VIM
11=set quantization matrix
12=reset audio interface after channel change or input switch (has no␣
↪argument).
   Needed for the cx2584x, not needed for the mspx4xx, but it doesn't seem␣
↪to
   do any harm calling it regardless.
13=set audio volume delay
14=set audio delay
```

Param[1]

Command value.

Decoder firmware API description

Note: this API is part of the decoder firmware, so it’s cx23415 only.

CX2341X_DEC_PING_FW

Enum: 0/0x00

Description

This API call does nothing. It may be used to check if the firmware is responding.

CX2341X_DEC_START_PLAYBACK

Enum: 1/0x01

Description

Begin or resume playback.

Param[0]

0 based frame number in GOP to begin playback from.

Param[1]

Specifies the number of muted audio frames to play before normal audio resumes. (This is not implemented in the firmware, leave at 0)

CX2341X_DEC_STOP_PLAYBACK

Enum: 2/0x02

Description

Ends playback and clears all decoder buffers. If PTS is not zero, playback stops at specified PTS.

Param[0]

Display 0=last frame, 1=black

Note: this takes effect immediately, so if you want to wait for a PTS, then use '0', otherwise the screen goes to black at once. You can call this later (even if there is no playback) with a 1 value to set the screen to black.

Param[1]

PTS low

Param[2]

PTS high

CX2341X_DEC_SET_PLAYBACK_SPEED

Enum: 3/0x03

Description

Playback stream at speed other than normal. There are two modes of operation:

- Smooth: host transfers entire stream and firmware drops unused frames.
- Coarse: host drops frames based on indexing as required to achieve desired speed.

Param[0]

```
Bitmap:
    0:7  0 normal
         1 fast only "1.5 times"
         n nX fast, 1/nX slow
    30   Framedrop:
             '0' during 1.5 times play, every other B frame is dropped
             '1' during 1.5 times play, stream is unchanged (bitrate
                 must not exceed 8mbps)
    31   Speed:
             '0' slow
             '1' fast
```

Note: n is limited to 2. Anything higher does not result in faster playback. Instead the host should start dropping frames.

Param[1]

Direction: 0=forward, 1=reverse

Note: to make reverse playback work you have to write full GOPs in reverse order.

Param[2]

```
Picture mask:
    1=I frames
    3=I, P frames
    7=I, P, B frames
```

Param[3]

B frames per GOP (for reverse play only)

Note: for reverse playback the Picture Mask should be set to I or I, P. Adding B frames to the mask will result in corrupt video. This field has to be set to the correct value in order to keep the timing correct.

Param[4]

Mute audio: 0=disable, 1=enable

Param[5]

Display 0=frame, 1=field

Param[6]

Specifies the number of muted audio frames to play before normal audio resumes. (Not implemented in the firmware, leave at 0)

CX2341X_DEC_STEP_VIDEO

Enum: 5/0x05

Description

Each call to this API steps the playback to the next unit defined below in the current playback direction.

Param[0]

0=frame, 1=top field, 2=bottom field

CX2341X_DEC_SET_DMA_BLOCK_SIZE

Enum: 8/0x08

Description

Set DMA transfer block size. Counterpart to API 0xC9

Param[0]

DMA transfer block size in bytes. A different size may be specified when issuing the DMA transfer command.

CX2341X_DEC_GET_XFER_INFO

Enum: 9/0x09

Description

This API call may be used to detect an end of stream condition.

Result[0]

Stream type

Result[1]

Address offset

Result[2]

Maximum bytes to transfer

Result[3]

Buffer fullness

CX2341X_DEC_GET_DMA_STATUS

Enum: 10/0x0A

Description

Status of the last DMA transfer

Result[0]

Bit 1 set means transfer complete Bit 2 set means DMA error Bit 3 set means linked list error

Result[1]

DMA type: 0=MPEG, 1=OSD, 2=YUV

CX2341X_DEC_SCHED_DMA_FROM_HOST

Enum: 11/0x0B

Description

Setup DMA from host operation. Counterpart to API 0xCC

Param[0]

Memory address of link list

Param[1]

Total # of bytes to transfer

Param[2]

DMA type (0=MPEG, 1=OSD, 2=YUV)

CX2341X_DEC_PAUSE_PLAYBACK

Enum: 13/0x0D

Description

Freeze playback immediately. In this mode, when internal buffers are full, no more data will be accepted and data request IRQs will be masked.

Param[0]

Display: 0=last frame, 1=black

CX2341X_DEC_HALT_FW

Enum: 14/0x0E

Description

The firmware is halted and no further API calls are serviced until the firmware is uploaded again.

CX2341X_DEC_SET_STANDARD

Enum: 16/0x10

Description

Selects display standard

Param[0]

0=NTSC, 1=PAL

CX2341X_DEC_GET_VERSION

Enum: 17/0x11

Description

Returns decoder firmware version information

Result[0]

Version bitmask:

- Bits 0:15 build
- Bits 16:23 minor
- Bits 24:31 major

CX2341X_DEC_SET_STREAM_INPUT

Enum: 20/0x14

Description

Select decoder stream input port

Param[0]

0=memory (default), 1=streaming

CX2341X_DEC_GET_TIMING_INFO

Enum: 21/0x15

Description

Returns timing information from start of playback

Result[0]

Frame count by decode order

Result[1]

Video PTS bits 0:31 by display order

Result[2]

Video PTS bit 32 by display order

Result[3]

SCR bits 0:31 by display order

Result[4]

SCR bit 32 by display order

CX2341X_DEC_SET_AUDIO_MODE

Enum: 22/0x16

Description

Select audio mode

Param[0]

Dual mono mode action 0=Stereo, 1=Left, 2=Right, 3=Mono, 4=Swap, -1=Unchanged

Param[1]

Stereo mode action: 0=Stereo, 1=Left, 2=Right, 3=Mono, 4=Swap, -1=Unchanged

CX2341X_DEC_SET_EVENT_NOTIFICATION

Enum: 23/0x17

Description

Setup firmware to notify the host about a particular event. Counterpart to API 0xD5

Param[0]

Event:

- 0=Audio mode change between mono, (joint) stereo and dual channel.
- 3=Decoder started
- 4=Unknown: goes off 10-15 times per second while decoding.
- 5=Some sync event: goes off once per frame.

Param[1]

Notification 0=disabled, 1=enabled

Param[2]

Interrupt bit

Param[3]

Mailbox slot, -1 if no mailbox required.

CX2341X_DEC_SET_DISPLAY_BUFFERS

Enum: 24/0x18

Description

Number of display buffers. To decode all frames in reverse playback you must use nine buffers.

Param[0]

0=six buffers, 1=nine buffers

CX2341X_DEC_EXTRACT_VBI

Enum: 25/0x19

Description

Extracts VBI data

Param[0]

0=extract from extension & user data, 1=extract from private packets

Result[0]

VBI table location

Result[1]

VBI table size

CX2341X_DEC_SET_DECODER_SOURCE

Enum: 26/0x1A

Description

Selects decoder source. Ensure that the parameters passed to this API match the encoder settings.

Param[0]

Mode: 0=MPEG from host, 1=YUV from encoder, 2=YUV from host

Param[1]

YUV picture width

Param[2]

YUV picture height

Param[3]

Bitmap: see Param[0] of API 0xBD

CX2341X_DEC_SET_PREBUFFERING

Enum: 30/0x1E

Description

Decoder prebuffering, when enabled up to 128KB are buffered for streams <8mpbs or 640KB for streams >8mbps

Param[0]

0=off, 1=on

PVR350 Video decoder registers 0x02002800 -> 0x02002B00

Author: Ian Armstrong <ian@iarmst.demon.co.uk>

Version: v0.4

Date: 12 March 2007

This list has been worked out through trial and error. There will be mistakes and omissions. Some registers have no obvious effect so it' s hard to say what they do, while others interact with each other, or require a certain load sequence. Horizontal filter setup is one example, with six registers working in unison and requiring a certain load sequence to correctly configure. The indexed colour palette is much easier to set at just two registers, but again it requires a certain load sequence.

Some registers are fussy about what they are set to. Load in a bad value & the decoder will fail. A firmware reload will often recover, but sometimes a reset is required. For registers containing size information, setting them to 0 is generally a bad idea. For other control registers i.e. 2878, you' ll only find out what values are bad when it hangs.

```
------------------------------------------------------------------------------
↪-----
2800
bit 0
        Decoder enable
        0 = disable
        1 = enable
------------------------------------------------------------------------------
↪-----
2804
bits 0:31
        Decoder horizontal Y alias register 1
---------------
2808
bits 0:31
        Decoder horizontal Y alias register 2
---------------
280C
bits 0:31
        Decoder horizontal Y alias register 3
---------------
2810
bits 0:31
```

(continues on next page)

(continued from previous page)

```
        Decoder horizontal Y alias register 4
---------------
2814
bits 0:31
        Decoder horizontal Y alias register 5
---------------
2818
bits 0:31
        Decoder horizontal Y alias trigger

These six registers control the horizontal aliasing filter for the Y plane.
The first five registers must all be loaded before accessing the trigger
(2818), as this register actually clocks the data through for the first
five.

To correctly program set the filter, this whole procedure must be done 16
times. The actual register contents are copied from a lookup-table in the
firmware which contains 4 different filter settings.

--------------------------------------------------------------------------
↪-----
281C
bits 0:31
        Decoder horizontal UV alias register 1
---------------
2820
bits 0:31
        Decoder horizontal UV alias register 2
---------------
2824
bits 0:31
        Decoder horizontal UV alias register 3
---------------
2828
bits 0:31
        Decoder horizontal UV alias register 4
---------------
282C
bits 0:31
        Decoder horizontal UV alias register 5
---------------
2830
bits 0:31
        Decoder horizontal UV alias trigger

These six registers control the horizontal aliasing for the UV plane.
Operation is the same as the Y filter, with 2830 being the trigger
register.

--------------------------------------------------------------------------
↪-----
2834
bits 0:15
        Decoder Y source width in pixels

bits 16:31
```

(continues on next page)

(continued from previous page)

```
        Decoder Y destination width in pixels
---------------
2838
bits 0:15
        Decoder UV source width in pixels

bits 16:31
        Decoder UV destination width in pixels

NOTE: For both registers, the resulting image must be fully visible on
screen. If the image exceeds the right edge both the source and destination
size must be adjusted to reflect the visible portion. For the source width,
you must take into account the scaling when calculating the new value.
-------------------------------------------------------------------------
↪-----

283C
bits 0:31
        Decoder Y horizontal scaling
                Normally = Reg 2854 >> 2
---------------
2840
bits 0:31
        Decoder ?? unknown - horizontal scaling
        Usually 0x00080514
---------------
2844
bits 0:31
        Decoder UV horizontal scaling
        Normally = Reg 2854 >> 2
---------------
2848
bits 0:31
        Decoder ?? unknown - horizontal scaling
        Usually 0x00100514
---------------
284C
bits 0:31
        Decoder ?? unknown - Y plane
        Usually 0x00200020
---------------
2850
bits 0:31
        Decoder ?? unknown - UV plane
        Usually 0x00200020
---------------
2854
bits 0:31
        Decoder 'master' value for horizontal scaling
---------------
2858
bits 0:31
        Decoder ?? unknown
        Usually 0
---------------
285C
```

(continues on next page)

(continued from previous page)

```
bits 0:31
        Decoder ?? unknown
        Normally = Reg 2854 >> 1
---------------
2860
bits 0:31
        Decoder ?? unknown
        Usually 0
---------------
2864
bits 0:31
        Decoder ?? unknown
        Normally = Reg 2854 >> 1
---------------
2868
bits 0:31
        Decoder ?? unknown
        Usually 0

Most of these registers either control horizontal scaling, or appear linked
to it in some way. Register 2854 contains the 'master' value & the other
registers can be calculated from that one. You must also remember to
correctly set the divider in Reg 2874.

To enlarge:
        Reg 2854 = (source_width * 0x00200000) / destination_width
        Reg 2874 = No divide

To reduce from full size down to half size:
        Reg 2854 = (source_width/2 * 0x00200000) / destination width
        Reg 2874 = Divide by 2

To reduce from half size down to quarter size:
        Reg 2854 = (source_width/4 * 0x00200000) / destination width
        Reg 2874 = Divide by 4

The result is always rounded up.

--------------------------------------------------------------------------------
↪-----
286C
bits 0:15
        Decoder horizontal Y buffer offset

bits 15:31
        Decoder horizontal UV buffer offset

Offset into the video image buffer. If the offset is gradually incremented,
the on screen image will move left & wrap around higher up on the right.

--------------------------------------------------------------------------------
↪-----
2870
bits 0:15
        Decoder horizontal Y output offset
```

(continues on next page)

(continued from previous page)

```
bits 16:31
        Decoder horizontal UV output offset

Offsets the actual video output. Controls output alignment of the Y & UV
planes. The higher the value, the greater the shift to the left. Use
reg 2890 to move the image right.

------------------------------------------------------------------------------
↪-----
2874
bits 0:1
        Decoder horizontal Y output size divider
        00 = No divide
        01 = Divide by 2
        10 = Divide by 3

bits 4:5
        Decoder horizontal UV output size divider
        00 = No divide
        01 = Divide by 2
        10 = Divide by 3

bit 8
        Decoder ?? unknown
        0 = Normal
        1 = Affects video output levels

bit 16
        Decoder ?? unknown
        0 = Normal
        1 = Disable horizontal filter

------------------------------------------------------------------------------
↪-----
2878
bit 0
        ?? unknown

bit 1
        osd on/off
        0 = osd off
        1 = osd on

bit 2
        Decoder + osd video timing
        0 = NTSC
        1 = PAL

bits 3:4
        ?? unknown

bit 5
        Decoder + osd
        Swaps upper & lower fields

------------------------------------------------------------------------------
↪-----
```

(continues on next page)

(continued from previous page)

```
287C
bits 0:10
        Decoder & osd ?? unknown
        Moves entire screen horizontally. Starts at 0x005 with the screen
        shifted heavily to the right. Incrementing in steps of 0x004 will
        gradually shift the screen to the left.

bits 11:31
        ?? unknown

Normally contents are 0x00101111 (NTSC) or 0x1010111d (PAL)

--------------------------------------------------------------------------
↪-----
2880  --------    ?? unknown
2884  --------    ?? unknown
--------------------------------------------------------------------------
↪-----
2888
bit 0
        Decoder + osd ?? unknown
        0 = Normal
        1 = Misaligned fields (Correctable through 289C & 28A4)

bit 4
        ?? unknown

bit 8
        ?? unknown

Warning: Bad values will require a firmware reload to recover.
                Known to be bad are 0x000,0x011,0x100,0x111
--------------------------------------------------------------------------
↪-----
288C
bits 0:15
        osd ?? unknown
        Appears to affect the osd position stability. The higher the value␣
↪the
        more unstable it becomes. Decoder output remains stable.

bits 16:31
        osd ?? unknown
        Same as bits 0:15

--------------------------------------------------------------------------
↪-----
2890
bits 0:11
        Decoder output horizontal offset.

Horizontal offset moves the video image right. A small left shift is
possible, but it's better to use reg 2870 for that due to its greater
range.

NOTE: Video corruption will occur if video window is shifted off the right
```

(continues on next page)

(continued from previous page)

```
edge. To avoid this read the notes for 2834 & 2838.
--------------------------------------------------------------------------------
↪-----
2894
bits 0:23
        Decoder output video surround colour.

Contains the colour (in yuv) used to fill the screen when the video is
running in a window.
--------------------------------------------------------------------------------
↪-----
2898
bits 0:23
        Decoder video window colour
        Contains the colour (in yuv) used to fill the video window when the
        video is turned off.

bit 24
        Decoder video output
        0 = Video on
        1 = Video off

bit 28
        Decoder plane order
        0 = Y,UV
        1 = UV,Y

bit 29
        Decoder second plane byte order
        0 = Normal (UV)
        1 = Swapped (VU)

In normal usage, the first plane is Y & the second plane is UV. Though the
order of the planes can be swapped, only the byte order of the second plane
can be swapped. This isn't much use for the Y plane, but can be useful for
the UV plane.

--------------------------------------------------------------------------------
↪-----
289C
bits 0:15
        Decoder vertical field offset 1

bits 16:31
        Decoder vertical field offset 2

Controls field output vertical alignment. The higher the number, the lower
the image on screen. Known starting values are 0x011E0017 (NTSC) &
0x01500017 (PAL)
--------------------------------------------------------------------------------
↪-----
28A0
bits 0:15
        Decoder & osd width in pixels

bits 16:31
```

(continues on next page)

(continued from previous page)

```
        Decoder & osd height in pixels

All output from the decoder & osd are disabled beyond this area. Decoder
output will simply go black outside of this region. If the osd tries to
exceed this area it will become corrupt.
------------------------------------------------------------------------
 ↪-----
28A4
bits 0:11
        osd left shift.

Has a range of 0x770->0x7FF. With the exception of 0, any value outside of
this range corrupts the osd.
------------------------------------------------------------------------
 ↪-----
28A8
bits 0:15
        osd vertical field offset 1

bits 16:31
        osd vertical field offset 2

Controls field output vertical alignment. The higher the number, the lower
the image on screen. Known starting values are 0x011E0017 (NTSC) &
0x01500017 (PAL)
------------------------------------------------------------------------
 ↪-----
28AC  --------    ?? unknown
|
V
28BC  --------    ?? unknown
------------------------------------------------------------------------
 ↪-----
28C0
bit 0
        Current output field
        0 = first field
        1 = second field

bits 16:31
        Current scanline
        The scanline counts from the top line of the first field
        through to the last line of the second field.
------------------------------------------------------------------------
 ↪-----
28C4  --------    ?? unknown
|
V
28F8  --------    ?? unknown
------------------------------------------------------------------------
 ↪-----
28FC
bit 0
        ?? unknown
        0 = Normal
        1 = Breaks decoder & osd output
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------------------------
↪-----
2900
bits 0:31
        Decoder vertical Y alias register 1
---------------
2904
bits 0:31
        Decoder vertical Y alias register 2
---------------
2908
bits 0:31
        Decoder vertical Y alias trigger

These three registers control the vertical aliasing filter for the Y plane.
Operation is similar to the horizontal Y filter (2804). The only real
difference is that there are only two registers to set before accessing
the trigger register (2908). As for the horizontal filter, the values are
taken from a lookup table in the firmware, and the procedure must be
repeated 16 times to fully program the filter.
------------------------------------------------------------------------------
↪-----
290C
bits 0:31
        Decoder vertical UV alias register 1
---------------
2910
bits 0:31
        Decoder vertical UV alias register 2
---------------
2914
bits 0:31
        Decoder vertical UV alias trigger

These three registers control the vertical aliasing filter for the UV
plane. Operation is the same as the Y filter, with 2914 being the trigger.
------------------------------------------------------------------------------
↪-----
2918
bits 0:15
        Decoder Y source height in pixels

bits 16:31
        Decoder Y destination height in pixels
---------------
291C
bits 0:15
        Decoder UV source height in pixels divided by 2

bits 16:31
        Decoder UV destination height in pixels

NOTE: For both registers, the resulting image must be fully visible on
screen. If the image exceeds the bottom edge both the source and
destination size must be adjusted to reflect the visible portion. For the
source height, you must take into account the scaling when calculating the
```

(continues on next page)

(continued from previous page)

```
new value.
--------------------------------------------------------------------------------
↪-----
2920
bits 0:31
        Decoder Y vertical scaling
        Normally = Reg 2930 >> 2
---------------
2924
bits 0:31
        Decoder Y vertical scaling
        Normally = Reg 2920 + 0x514
---------------
2928
bits 0:31
        Decoder UV vertical scaling
        When enlarging = Reg 2930 >> 2
        When reducing = Reg 2930 >> 3
---------------
292C
bits 0:31
        Decoder UV vertical scaling
        Normally = Reg 2928 + 0x514
---------------
2930
bits 0:31
        Decoder 'master' value for vertical scaling
---------------
2934
bits 0:31
        Decoder ?? unknown - Y vertical scaling
---------------
2938
bits 0:31
        Decoder Y vertical scaling
        Normally = Reg 2930
---------------
293C
bits 0:31
        Decoder ?? unknown - Y vertical scaling
---------------
2940
bits 0:31
        Decoder UV vertical scaling
        When enlarging = Reg 2930 >> 1
        When reducing = Reg 2930
---------------
2944
bits 0:31
        Decoder ?? unknown - UV vertical scaling
---------------
2948
bits 0:31
        Decoder UV vertical scaling
        Normally = Reg 2940
---------------
```

(continues on next page)

(continued from previous page)

```
294C
bits 0:31
        Decoder ?? unknown - UV vertical scaling

Most of these registers either control vertical scaling, or appear linked
to it in some way. Register 2930 contains the 'master' value & all other
registers can be calculated from that one. You must also remember to
correctly set the divider in Reg 296C

To enlarge:
        Reg 2930 = (source_height * 0x00200000) / destination_height
        Reg 296C = No divide

To reduce from full size down to half size:
        Reg 2930 = (source_height/2 * 0x00200000) / destination height
        Reg 296C = Divide by 2

To reduce from half down to quarter.
        Reg 2930 = (source_height/4 * 0x00200000) / destination height
        Reg 296C = Divide by 4

------------------------------------------------------------------------------
↪-----
2950
bits 0:15
        Decoder Y line index into display buffer, first field

bits 16:31
        Decoder Y vertical line skip, first field
------------------------------------------------------------------------------
↪-----
2954
bits 0:15
        Decoder Y line index into display buffer, second field

bits 16:31
        Decoder Y vertical line skip, second field
------------------------------------------------------------------------------
↪-----
2958
bits 0:15
        Decoder UV line index into display buffer, first field

bits 16:31
        Decoder UV vertical line skip, first field
------------------------------------------------------------------------------
↪-----
295C
bits 0:15
        Decoder UV line index into display buffer, second field

bits 16:31
        Decoder UV vertical line skip, second field
------------------------------------------------------------------------------
↪-----
2960
```

(continues on next page)

(continued from previous page)

```
bits 0:15
        Decoder destination height minus 1

bits 16:31
        Decoder destination height divided by 2
------------------------------------------------------------------------------
↪-----
2964
bits 0:15
        Decoder Y vertical offset, second field

bits 16:31
        Decoder Y vertical offset, first field

These two registers shift the Y plane up. The higher the number, the
greater the shift.
------------------------------------------------------------------------------
↪-----
2968
bits 0:15
        Decoder UV vertical offset, second field

bits 16:31
        Decoder UV vertical offset, first field

These two registers shift the UV plane up. The higher the number, the
greater the shift.
------------------------------------------------------------------------------
↪-----
296C
bits 0:1
        Decoder vertical Y output size divider
        00 = No divide
        01 = Divide by 2
        10 = Divide by 4

bits 8:9
        Decoder vertical UV output size divider
        00 = No divide
        01 = Divide by 2
        10 = Divide by 4
------------------------------------------------------------------------------
↪-----
2970
bit 0
        Decoder ?? unknown
        0 = Normal
        1 = Affect video output levels

bit 16
        Decoder ?? unknown
        0 = Normal
        1 = Disable vertical filter

------------------------------------------------------------------------------
↪-----
```

(continues on next page)

(continued from previous page)

```
2974  --------   ?? unknown
|
V
29EF  --------   ?? unknown
------------------------------------------------------------------------------
↪-----
2A00
bits 0:2
        osd colour mode
        000 = 8 bit indexed
        001 = 16 bit (565)
        010 = 15 bit (555)
        011 = 12 bit (444)
        100 = 32 bit (8888)

bits 4:5
        osd display bpp
        01 = 8 bit
        10 = 16 bit
        11 = 32 bit

bit 8
        osd global alpha
        0 = Off
        1 = On

bit 9
        osd local alpha
        0 = Off
        1 = On

bit 10
        osd colour key
        0 = Off
        1 = On

bit 11
        osd ?? unknown
        Must be 1

bit 13
        osd colour space
        0 = ARGB
        1 = AYVU

bits 16:31
        osd ?? unknown
        Must be 0x001B (some kind of buffer pointer ?)

When the bits-per-pixel is set to 8, the colour mode is ignored and
assumed to be 8 bit indexed. For 16 & 32 bits-per-pixel the colour depth
is honoured, and when using a colour depth that requires fewer bytes than
allocated the extra bytes are used as padding. So for a 32 bpp with 8 bit
index colour, there are 3 padding bytes per pixel. It's also possible to
select 16bpp with a 32 bit colour mode. This results in the pixel width
being doubled, but the color key will not work as expected in this mode.
```

(continues on next page)

(continued from previous page)

```
Colour key is as it suggests. You designate a colour which will become
completely transparent. When using 565, 555 or 444 colour modes, the
colour key is always 16 bits wide. The colour to key on is set in Reg 2A18.

Local alpha works differently depending on the colour mode. For 32bpp & 8
bit indexed, local alpha is a per-pixel 256 step transparency, with 0 being
transparent and 255 being solid. For the 16bpp modes 555 & 444, the unused
bit(s) act as a simple transparency switch, with 0 being solid & 1 being
fully transparent. There is no local alpha support for 16bit 565.

Global alpha is a 256 step transparency that applies to the entire osd,
with 0 being transparent & 255 being solid.

It's possible to combine colour key, local alpha & global alpha.
--------------------------------------------------------------------------
↪-----
2A04
bits 0:15
        osd x coord for left edge

bits 16:31
        osd y coord for top edge
---------------
2A08
bits 0:15
        osd x coord for right edge

bits 16:31
        osd y coord for bottom edge

For both registers, (0,0) = top left corner of the display area. These
registers do not control the osd size, only where it's positioned & how
much is visible. The visible osd area cannot exceed the right edge of the
display, otherwise the osd will become corrupt. See reg 2A10 for
setting osd width.
--------------------------------------------------------------------------
↪-----
2A0C
bits 0:31
        osd buffer index

An index into the osd buffer. Slowly incrementing this moves the osd left,
wrapping around onto the right edge
--------------------------------------------------------------------------
↪-----
2A10
bits 0:11
        osd buffer 32 bit word width

Contains the width of the osd measured in 32 bit words. This means that all
colour modes are restricted to a byte width which is divisible by 4.
--------------------------------------------------------------------------
↪-----
2A14
bits 0:15
```

(continues on next page)

(continued from previous page)

```
        osd height in pixels

bits 16:32
        osd line index into buffer
        osd will start displaying from this line.
-------------------------------------------------------------------------
 ↪-----
2A18
bits 0:31
        osd colour key

Contains the colour value which will be transparent.
-------------------------------------------------------------------------
 ↪-----
2A1C
bits 0:7
        osd global alpha

Contains the global alpha value (equiv ivtvfbctl --alpha XX)
-------------------------------------------------------------------------
 ↪-----
2A20  --------    ?? unknown
|
V
2A2C  --------    ?? unknown
-------------------------------------------------------------------------
 ↪-----
2A30
bits 0:7
        osd colour to change in indexed palette
---------------
2A34
bits 0:31
        osd colour for indexed palette

To set the new palette, first load the index of the colour to change into
2A30, then load the new colour into 2A34. The full palette is 256 colours,
so the index range is 0x00-0xFF
-------------------------------------------------------------------------
 ↪-----
2A38  --------    ?? unknown
2A3C  --------    ?? unknown
-------------------------------------------------------------------------
 ↪-----
2A40
bits 0:31
        osd ?? unknown

Affects overall brightness, wrapping around to black
-------------------------------------------------------------------------
 ↪-----
2A44
bits 0:31
        osd ?? unknown

Green tint
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------------------------
↪-----
2A48
bits 0:31
        osd ?? unknown

Red tint
------------------------------------------------------------------------------
↪-----
2A4C
bits 0:31
        osd ?? unknown

Affects overall brightness, wrapping around to black
------------------------------------------------------------------------------
↪-----
2A50
bits 0:31
        osd ?? unknown

Colour shift
------------------------------------------------------------------------------
↪-----
2A54
bits 0:31
        osd ?? unknown

Colour shift
------------------------------------------------------------------------------
↪-----
2A58  --------    ?? unknown
|
V
2AFC  --------    ?? unknown
------------------------------------------------------------------------------
↪-----
2B00
bit 0
        osd filter control
        0 = filter off
        1 = filter on

bits 1:4
        osd ?? unknown

------------------------------------------------------------------------------
↪-----
```

The cx231xx DMA engine

This page describes the structures and procedures used by the cx2341x DMA engine.

Introduction

The cx2341x PCI interface is busmaster capable. This means it has a DMA engine to efficiently transfer large volumes of data between the card and main memory without requiring help from a CPU. Like most hardware, it must operate on contiguous physical memory. This is difficult to come by in large quantities on virtual memory machines.

Therefore, it also supports a technique called “scatter-gather”. The card can transfer multiple buffers in one operation. Instead of allocating one large contiguous buffer, the driver can allocate several smaller buffers.

In practice, I’ve seen the average transfer to be roughly 80K, but transfers above 128K were not uncommon, particularly at startup. The 128K figure is important, because that is the largest block that the kernel can normally allocate. Even still, 128K blocks are hard to come by, so the driver writer is urged to choose a smaller block size and learn the scatter-gather technique.

Mailbox #10 is reserved for DMA transfer information.

Note: the hardware expects little-endian data (‘intel format’).

Flow

This section describes, in general, the order of events when handling DMA transfers. Detailed information follows this section.

- The card raises the Encoder interrupt.
- The driver reads the transfer type, offset and size from Mailbox #10.
- The driver constructs the scatter-gather array from enough free dma buffers to cover the size.
- The driver schedules the DMA transfer via the ScheduleDMAtoHost API call.
- The card raises the DMA Complete interrupt.
- The driver checks the DMA status register for any errors.
- The driver post-processes the newly transferred buffers.

NOTE! It is possible that the Encoder and DMA Complete interrupts get raised simultaneously. (End of the last, start of the next, etc.)

Mailbox #10

The Flags, Command, Return Value and Timeout fields are ignored.

- Name: Mailbox #10
- Results[0]: Type: 0: MPEG.
- Results[1]: Offset: The position relative to the card’s memory space.
- Results[2]: Size: The exact number of bytes to transfer.

My speculation is that since the StartCapture API has a capture type of “RAW” available, that the type field will have other values that correspond to YUV and PCM data.

Scatter-Gather Array

The scatter-gather array is a contiguously allocated block of memory that tells the card the source and destination of each data-block to transfer. Card “addresses”are derived from the offset supplied by Mailbox #10. Host addresses are the physical memory location of the target DMA buffer.

Each S-G array element is a struct of three 32-bit words. The first word is the source address, the second is the destination address. Both take up the entire 32 bits. The lowest 18 bits of the third word is the transfer byte count. The high-bit of the third word is the “last”flag. The last-flag tells the card to raise the DMA_DONE interrupt. From hard personal experience, if you forget to set this bit, the card will still “work” but the stream will most likely get corrupted.

The transfer count must be a multiple of 256. Therefore, the driver will need to track how much data in the target buffer is valid and deal with it accordingly.

Array Element:

- 32-bit Source Address
- 32-bit Destination Address
- 14-bit reserved (high bit is the last flag)
- 18-bit byte count

DMA Transfer Status

Register 0x0004 holds the DMA Transfer Status:

- bit 0: read completed
- bit 1: write completed
- bit 2: DMA read error
- bit 3: DMA write error
- bit 4: Scatter-Gather array error

The cx88 driver

Author: Gerd Hoffmann

Documentation missing at the cx88 datasheet

MO_OUTPUT_FORMAT (0x310164)

```
Previous default from DScaler: 0x1c1f0008
Digit 8: 31-28
28: PREVREMOD = 1

Digit 7: 27-24 (0xc = 12 = b1100 )
27: COMBALT = 1
26: PAL_INV_PHASE
  (DScaler apparently set this to 1, resulted in sucky picture)

Digits 6,5: 23-16
25-16: COMB_RANGE = 0x1f [default] (9 bits -> max 512)

Digit 4: 15-12
15: DISIFX = 0
14: INVCBF = 0
13: DISADAPT = 0
12: NARROWADAPT = 0

Digit 3: 11-8
11: FORCE2H
10: FORCEREMD
9: NCHROMAEN
8: NREMODEN

Digit 2: 7-4
7-6: YCORE
5-4: CCORE

Digit 1: 3-0
3: RANGE = 1
2: HACTEXT
1: HSFMT
```

0x47 is the sync byte for MPEG-2 transport stream packets. Datasheet incorrectly states to use 47 decimal. 188 is the length. All DVB compliant frontends output packets with this start code.

Hauppauge WinTV cx88 IR information

The controls for the mux are GPIO [0,1] for source, and GPIO 2 for muting.

GPIO0	GPIO1	
0	0	TV Audio
1	0	FM radio
0	1	Line-In
1	1	Mono tuner bypass or CD passthru (tuner specific)

GPIO 16(I believe) is tied to the IR port (if present).

From the data sheet:

- Register 24’h20004 PCI Interrupt Status
- bit [18] IR_SMP_INT Set when 32 input samples have been collected over
- gpio[16] pin into GP_SAMPLE register.

What’s missing from the data sheet:

- Setup 4KHz sampling rate (roughly 2x oversampled; good enough for our RC5 compat remote)
- set register 0x35C050 to 0xa80a80
- enable sampling
- set register 0x35C054 to 0x5
- enable the IRQ bit 18 in the interrupt mask register (and provide for a handler)

GP_SAMPLE register is at 0x35C058

Bits are then right shifted into the GP_SAMPLE register at the specified rate; you get an interrupt when a full DWORD is received. You need to recover the actual RC5 bits out of the (oversampled) IR sensor bits. (Hint: look for the 0/1and 1/0 crossings of the RC5 bi-phase data) An actual raw RC5 code will span 2-3 DWORDS, depending on the actual alignment.

I’m pretty sure when no IR signal is present the receiver is always in a marking state(1); but stray light, etc can cause intermittent noise values as well. Remember, this is a free running sample of the IR receiver state over time, so don’t assume any sample starts at any particular place.

Additional info

This data sheet (google search) seems to have a lovely description of the RC5 basics: http://www.atmel.com/dyn/resources/prod_documents/doc2817.pdf

This document has more data: http://www.nenya.be/beor/electronics/rc5.htm

This document has a how to decode a bi-phase data stream: http://www.ee.washington.edu/circuit_archive/text/ir_decode.txt

This document has still more info: http://www.xs4all.nl/~sbp/knowledge/ir/rc5.htm

The VPBE V4L2 driver design

File partitioning

V4L2 display device driver drivers/media/platform/davinci/vpbe_display.c
drivers/media/platform/davinci/vpbe_display.h

VPBE display controller drivers/media/platform/davinci/vpbe.c
drivers/media/platform/davinci/vpbe.h

VPBE venc sub device driver drivers/media/platform/davinci/vpbe_venc.c
drivers/media/platform/davinci/vpbe_venc.h
drivers/media/platform/davinci/vpbe_venc_regs.h

VPBE osd driver drivers/media/platform/davinci/vpbe_osd.c
drivers/media/platform/davinci/vpbe_osd.h
drivers/media/platform/davinci/vpbe_osd_regs.h

To be done

vpbe display controller

- Add support for external encoders.
- add support for selecting external encoder as default at probe time.

vpbe venc sub device

- add timings for supporting ths8200
- add support for LogicPD LCD.

FB drivers

- Add support for fbdev drivers.- Ready and part of subsequent patches.

The Samsung S5P/EXYNOS4 FIMC driver

Files partitioning

- media device driver

  drivers/media/platform/exynos4-is/media-dev.[ch]

- camera capture video device driver

  drivers/media/platform/exynos4-is/fimc-capture.c

- MIPI-CSI2 receiver subdev

  drivers/media/platform/exynos4-is/mipi-csis.[ch]

- video post-processor (mem-to-mem)

  drivers/media/platform/exynos4-is/fimc-core.c

- common files

  drivers/media/platform/exynos4-is/fimc-core.h drivers/media/platform/exynos4-is/fimc-reg.h drivers/media/platform/exynos4-is/regs-fimc.h

The pvrusb2 driver

Author: Mike Isely <isely@pobox.com>

Background

This driver is intended for the "Hauppauge WinTV PVR USB 2.0", which is a USB 2.0 hosted TV Tuner. This driver is a work in progress. Its history started with the reverse-engineering effort by Björn Danielsson <pvrusb2@dax.nu> whose web page can be found here: http://pvrusb2.dax.nu/

From there Aurelien Alleaume <slts@free.fr> began an effort to create a video4linux compatible driver. I began with Aurelien's last known snapshot and evolved the driver to the state it is in here.

More information on this driver can be found at: http://www.isely.net/pvrusb2.html

This driver has a strong separation of layers. They are very roughly:

1. Low level wire-protocol implementation with the device.
2. I2C adaptor implementation and corresponding I2C client drivers implemented elsewhere in V4L.
3. High level hardware driver implementation which coordinates all activities that ensure correct operation of the device.

4. A "context" layer which manages instancing of driver, setup, tear-down, arbitration, and interaction with high level interfaces appropriately as devices are hotplugged in the system.
5. High level interfaces which glue the driver to various published Linux APIs (V4L, sysfs, maybe DVB in the future).

The most important shearing layer is between the top 2 layers. A lot of work went into the driver to ensure that any kind of conceivable API can be laid on top of the core driver. (Yes, the driver internally leverages V4L to do its work but that really has nothing to do with the API published by the driver to the outside world.) The architecture allows for different APIs to simultaneously access the driver. I have a strong sense of fairness about APIs and also feel that it is a good design principle to keep implementation and interface isolated from each other. Thus while right now the V4L high level interface is the most complete, the sysfs high level interface will work equally well for similar functions, and there's no reason I see right now why it shouldn't be possible to produce a DVB high level interface that can sit right alongside V4L.

Building

To build these modules essentially amounts to just running "Make", but you need the kernel source tree nearby and you will likely also want to set a few controlling environment variables first in order to link things up with that source tree. Please see the Makefile here for comments that explain how to do that.

Source file list / functional overview

(Note: The term "module" used below generally refers to loosely defined functional units within the pvrusb2 driver and bears no relation to the Linux kernel's concept of a loadable module.)

pvrusb2-audio.[ch] - This is glue logic that resides between this driver and the msp3400.ko I2C client driver (which is found elsewhere in V4L).

pvrusb2-context.[ch] - This module implements the context for an instance of the driver. Everything else eventually ties back to or is otherwise instanced within the data structures implemented here. Hotplugging is ultimately coordinated here. All high level interfaces tie into the driver through this module. This module helps arbitrate each interface's access to the actual driver core, and is designed to allow concurrent access through multiple instances of multiple interfaces (thus you can for example change the tuner's frequency through sysfs while simultaneously streaming video through V4L out to an instance of mplayer).

pvrusb2-debug.h - This header defines a printk() wrapper and a mask of debugging bit definitions for the various kinds of debug messages that can be enabled within the driver.

pvrusb2-debugifc.[ch] - This module implements a crude command line oriented debug interface into the driver. Aside from being part of the process for implementing manual firmware extraction (see the pvrusb2 web

site mentioned earlier), probably I' m the only one who has ever used this. It is mainly a debugging aid.

pvrusb2-eeprom.[ch] - This is glue logic that resides between this driver the tveeprom.ko module, which is itself implemented elsewhere in V4L.

pvrusb2-encoder.[ch] - This module implements all protocol needed to interact with the Conexant mpeg2 encoder chip within the pvrusb2 device. It is a crude echo of corresponding logic in ivtv, however the design goals (strict isolation) and physical layer (proxy through USB instead of PCI) are enough different that this implementation had to be completely different.

pvrusb2-hdw-internal.h - This header defines the core data structure in the driver used to track ALL internal state related to control of the hardware. Nobody outside of the core hardware-handling modules should have any business using this header. All external access to the driver should be through one of the high level interfaces (e.g. V4L, sysfs, etc), and in fact even those high level interfaces are restricted to the API defined in pvrusb2-hdw.h and NOT this header.

pvrusb2-hdw.h - This header defines the full internal API for controlling the hardware. High level interfaces (e.g. V4L, sysfs) will work through here.

pvrusb2-hdw.c - This module implements all the various bits of logic that handle overall control of a specific pvrusb2 device. (Policy, instantiation, and arbitration of pvrusb2 devices fall within the jurisdiction of pvrusb-context not here).

pvrusb2-i2c-chips-*.c - These modules implement the glue logic to tie together and configure various I2C modules as they attach to the I2C bus. There are two versions of this file. The "v4l2" version is intended to be used in-tree alongside V4L, where we implement just the logic that makes sense for a pure V4L environment. The "all" version is intended for use outside of V4L, where we might encounter other possibly "challenging" modules from ivtv or older kernel snapshots (or even the support modules in the standalone snapshot).

pvrusb2-i2c-cmd-v4l1.[ch] - This module implements generic V4L1 compatible commands to the I2C modules. It is here where state changes inside the pvrusb2 driver are translated into V4L1 commands that are in turn send to the various I2C modules.

pvrusb2-i2c-cmd-v4l2.[ch] - This module implements generic V4L2 compatible commands to the I2C modules. It is here where state changes inside the pvrusb2 driver are translated into V4L2 commands that are in turn send to the various I2C modules.

pvrusb2-i2c-core.[ch] - This module provides an implementation of a kernel-friendly I2C adaptor driver, through which other external I2C client drivers (e.g. msp3400, tuner, lirc) may connect and operate corresponding chips within the pvrusb2 device. It is through here that other V4L modules can reach into this driver to operate specific pieces (and those modules are in turn driven by glue logic which is coordinated by pvrusb2-hdw, doled out by pvrusb2-context, and then ultimately made available to users through one of the high level interfaces).

pvrusb2-io.[ch] - This module implements a very low level ring of transfer

buffers, required in order to stream data from the device. This module is very low level. It only operates the buffers and makes no attempt to define any policy or mechanism for how such buffers might be used.

pvrusb2-ioread.[ch] - This module layers on top of pvrusb2-io.[ch] to provide a streaming API usable by a read() system call style of I/O. Right now this is the only layer on top of pvrusb2-io.[ch], however the underlying architecture here was intended to allow for other styles of I/O to be implemented with additional modules, like mmap()'ed buffers or something even more exotic.

pvrusb2-main.c - This is the top level of the driver. Module level and USB core entry points are here. This is our "main".

pvrusb2-sysfs.[ch] - This is the high level interface which ties the pvrusb2 driver into sysfs. Through this interface you can do everything with the driver except actually stream data.

pvrusb2-tuner.[ch] - This is glue logic that resides between this driver and the tuner.ko I2C client driver (which is found elsewhere in V4L).

pvrusb2-util.h - This header defines some common macros used throughout the driver. These macros are not really specific to the driver, but they had to go somewhere.

pvrusb2-v4l2.[ch] - This is the high level interface which ties the pvrusb2 driver into video4linux. It is through here that V4L applications can open and operate the driver in the usual V4L ways. Note that **ALL** V4L functionality is published only through here and nowhere else.

pvrusb2-video-*.[ch] - This is glue logic that resides between this driver and the saa711x.ko I2C client driver (which is found elsewhere in V4L). Note that saa711x.ko used to be known as saa7115.ko in ivtv. There are two versions of this; one is selected depending on the particular saa711[5x].ko that is found.

pvrusb2.h - This header contains compile time tunable parameters (and at the moment the driver has very little that needs to be tuned).

PXA-Camera Host Driver

Author: Robert Jarzmik <robert.jarzmik@free.fr>

Constraints

a) Image size for YUV422P format All YUV422P images are enforced to have width x height % 16 = 0. This is due to DMA constraints, which transfers only planes of 8 byte multiples.

Global video workflow

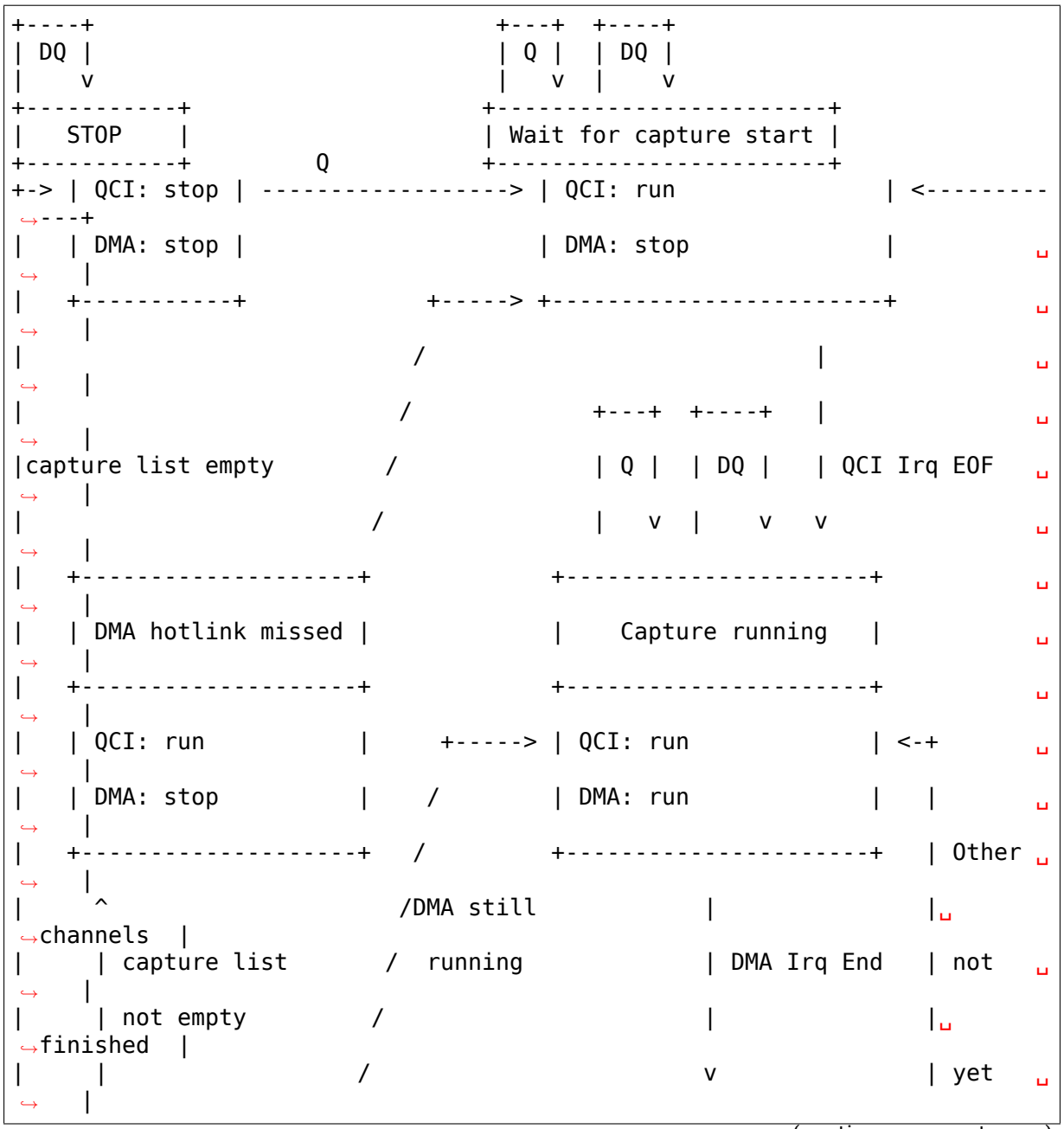
a) QCI stopped Initially, the QCI interface is stopped. When a buffer is queued (pxa_videobuf_ops->buf_queue), the QCI starts.

b) QCI started More buffers can be queued while the QCI is started without halting the capture. The new buffers are "appended" at the tail of the DMA chain, and smoothly captured one frame after the other.

Once a buffer is filled in the QCI interface, it is marked as "DONE" and removed from the active buffers list. It can be then requeud or dequeued by userland application.

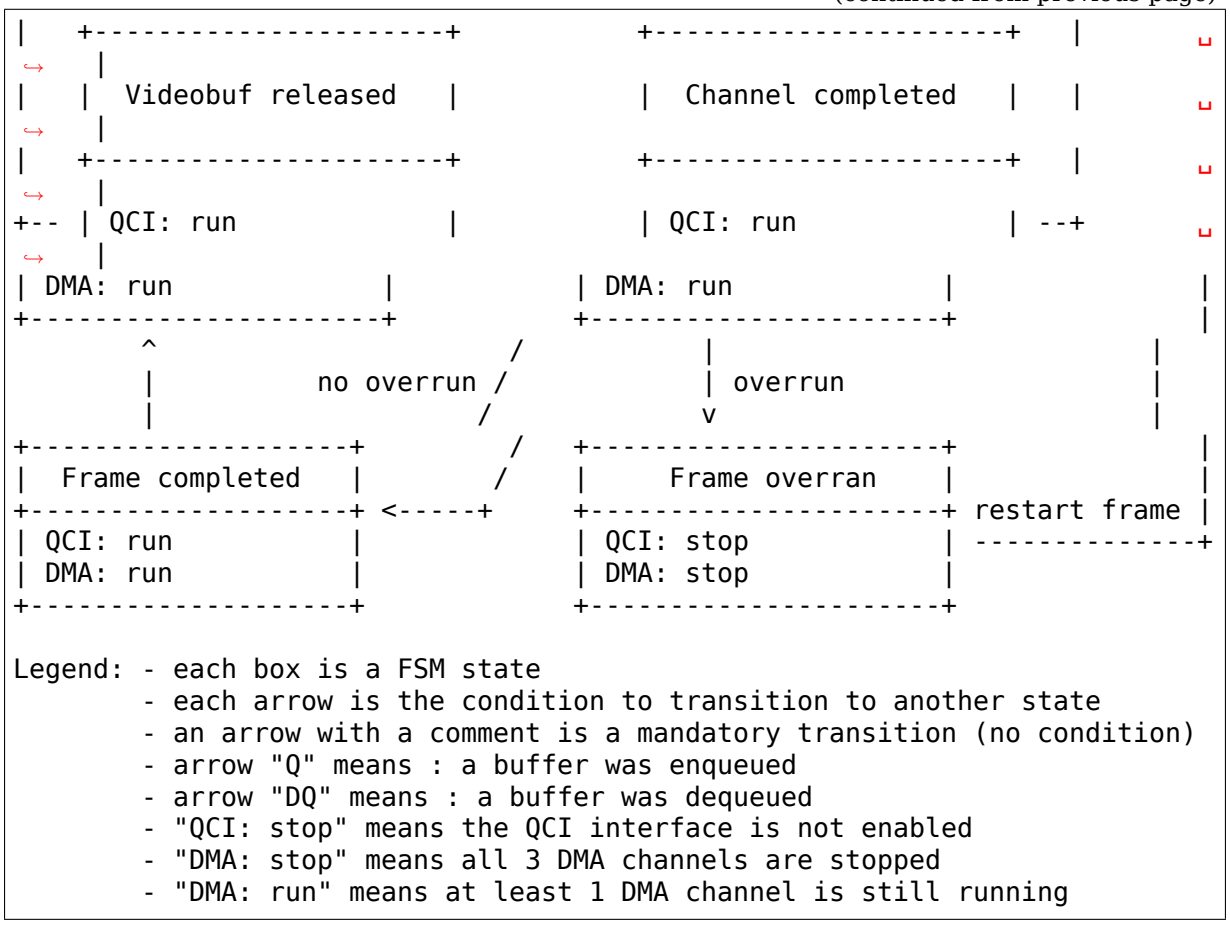
Once the last buffer is filled in, the QCI interface stops.

c) Capture global finite state machine schema



(continues on next page)

(continued from previous page)

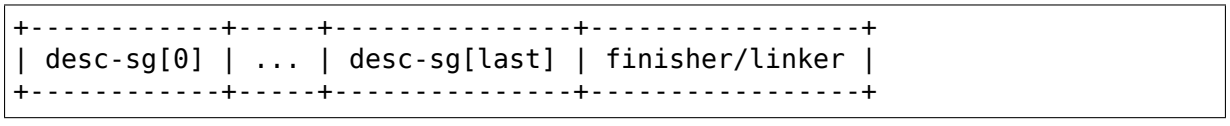


DMA usage

a) **DMA flow**

- first buffer queued for capture Once a first buffer is queued for capture, the QCI is started, but data transfer is not started. On "End Of Frame" interrupt, the irq handler starts the DMA chain.
- capture of one videobuffer The DMA chain starts transferring data into videobuffer RAM pages. When all pages are transferred, the DMA irq is raised on "ENDINTR" status
- finishing one videobuffer The DMA irq handler marks the videobuffer as "done", and removes it from the active running queue Meanwhile, the next videobuffer (if there is one), is transferred by DMA
- finishing the last videobuffer On the DMA irq of the last videobuffer, the QCI is stopped.

b) DMA prepared buffer will have this structure



This structure is pointed by dma->sg_cpu. The descriptors are used as follows:

- desc-sg[i]: i-th descriptor, transferring the i-th sg element to the video buffer scatter gather
- finisher: has ddadr=DADDR_STOP, dcmd=ENDIRQEN
- linker: has ddadr= desc-sg[0] of next video buffer, dcmd=0

For the next schema, let's assume d0=desc-sg[0] .. dN=desc-sg[N], "f" stands for finisher and "l" for linker. A typical running chain is :

```
    Videobuffer 1          Videobuffer 2
+---------+----+---+  +----+----+----+---+
| d0 | .. | dN | l |  | d0 | .. | dN | f |
+---------+----+-|-+  ^----+----+----+---+
                 |    |
                 +----+
```

After the chaining is finished, the chain looks like :

```
    Videobuffer 1          Videobuffer 2          Videobuffer 3
+---------+----+---+  +----+----+----+---+  +----+----+----+---+
| d0 | .. | dN | l |  | d0 | .. | dN | l |  | d0 | .. | dN | f |
+---------+----+-|-+  ^----+----+----+-|-+  ^----+----+----+---+
                 |    |                |    |
                 +----+                +----+
                                     new_link
```

c) DMA hot chaining timeslice issue

As DMA chaining is done while DMA _is_ running, the linking may be done while the DMA jumps from one Videobuffer to another. On the schema, that would be a problem if the following sequence is encountered :

- DMA chain is Videobuffer1 + Videobuffer2
- pxa_videobuf_queue() is called to queue Videobuffer3
- DMA controller finishes Videobuffer2, and DMA stops

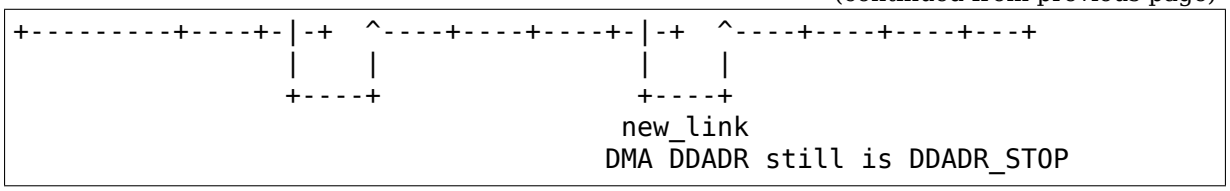
```
 =>
    Videobuffer 1          Videobuffer 2
+---------+----+---+  +----+----+----+---+
| d0 | .. | dN | l |  | d0 | .. | dN | f |
+---------+----+-|-+  ^----+----+----+-^-+
                 |    |                |
                 +----+                +-- DMA DDADR loads DDADR_STOP
```

- pxa_dma_add_tail_buf() is called, the Videobuffer2 "finisher" is replaced by a "linker" to Videobuffer3 (creation of new_link)
- pxa_videobuf_queue() finishes
- the DMA irq handler is called, which terminates Videobuffer2
- Videobuffer3 capture is not scheduled on DMA chain (as it stopped !!!)

```
    Videobuffer 1          Videobuffer 2          Videobuffer 3
+---------+----+---+  +----+----+----+---+  +----+----+----+---+
| d0 | .. | dN | l |  | d0 | .. | dN | l |  | d0 | .. | dN | f |
```

(continues on next page)

(continued from previous page)



- pxa_camera_check_link_miss() is called This checks if the DMA is finished and a buffer is still on the pcdev->capture list. If that's the case, the capture will be restarted, and Videobuffer3 is scheduled on DMA chain.
- the DMA irq handler finishes

Note: If DMA stops just after pxa_camera_check_link_miss() reads DDADR() value, we have the guarantee that the DMA irq handler will be called back when the DMA will finish the buffer, and pxa_camera_check_link_miss() will be called again, to reschedule Videobuffer3.

The Radiotrack radio driver

Author: Stephen M. Benoit <benoits@servicepro.com>

Date: Dec 14, 1996

ACKNOWLEDGMENTS

This document was made based on 'C' code for Linux from Gideon le Grange (legrang@active.co.za or legrang@cs.sun.ac.za) in 1994, and elaborations from Frans Brinkman (brinkman@esd.nl) in 1996. The results reported here are from experiments that the author performed on his own setup, so your mileage may vary ···I make no guarantees, claims or warranties to the suitability or validity of this information. No other documentation on the AIMS Lab (http://www.aimslab.com/) RadioTrack card was made available to the author. This document is offered in the hopes that it might help users who want to use the RadioTrack card in an environment other than MS Windows.

WHY THIS DOCUMENT?

I have a RadioTrack card from back when I ran an MS-Windows platform. After converting to Linux, I found Gideon le Grange's command-line software for running the card, and found that it was good! Frans Brinkman made a comfortable X-windows interface, and added a scanning feature. For hack value, I wanted to see if the tuner could be tuned beyond the usual FM radio broadcast band, so I could pick up the audio carriers from North American broadcast TV channels, situated just below and above the 87.0-109.0 MHz range. I did not get much success, but I learned about programming ioports under Linux and gained some insights about the hardware design used for the card.

So, without further delay, here are the details.

PHYSICAL DESCRIPTION

The RadioTrack card is an ISA 8-bit FM radio card. The radio frequency (RF) input is simply an antenna lead, and the output is a power audio signal available through a miniature phone plug. Its RF frequencies of operation are more or less limited from 87.0 to 109.0 MHz (the commercial FM broadcast band). Although the registers can be programmed to request frequencies beyond these limits, experiments did not give promising results. The variable frequency oscillator (VFO) that demodulates the intermediate frequency (IF) signal probably has a small range of useful frequencies, and wraps around or gets clipped beyond the limits mentioned above.

CONTROLLING THE CARD WITH IOPORT

The RadioTrack (base) ioport is configurable for 0x30c or 0x20c. Only one ioport seems to be involved. The ioport decoding circuitry must be pretty simple, as individual ioport bits are directly matched to specific functions (or blocks) of the radio card. This way, many functions can be changed in parallel with one write to the ioport. The only feedback available through the ioports appears to be the "Stereo Detect" bit.

The bits of the ioport are arranged as follows:

```
MSb                                                             LSb
+------+------+------+--------+--------+-------+---------+--------+
| VolA | VolB | ???? | Stereo | Radio  | TuneA | TuneB   | Tune   |
|  (+) |  (-) |      | Detect | Audio  | (bit) | (latch) | Update |
|      |      |      | Enable | Enable |       |         | Enable |
+------+------+------+--------+--------+-------+---------+--------+
```

VolA	VolB	Description
0	0	audio mute
0	1	volume + (some delay required)
1	0	volume - (some delay required)
1	1	stay at present volume

Stereo Detect Enable	Description
0	No Detect
1	Detect

Results available by reading ioport >60 msec after last port write.

0xff ==> no stereo detected, 0xfd ==> stereo detected.

Radio to Audio (path) Enable	Description
0	Disable path (silence)
1	Enable path (audio produced)

TuneA	TuneB	Description
0	0	“zero” bit phase 1
0	1	“zero” bit phase 2
1	0	“one” bit phase 1
1	1	“one” bit phase 2

24-bit code, where bits = (freq*40) + 10486188. The Most Significant 11 bits must be 1010 xxxx 0x0 to be valid. The bits are shifted in LSb first.

Tune Update Enable	Description
0	Tuner held constant
1	Tuner updating in progress

PROGRAMMING EXAMPLES

```
Default:         BASE <-- 0xc8  (current volume, no stereo detect,
                                 radio enable, tuner adjust disable)

Card Off:        BASE <-- 0x00  (audio mute, no stereo detect,
                                 radio disable, tuner adjust disable)

Card On:         BASE <-- 0x00  (see "Card Off", clears any unfinished␣
↪business)
                 BASE <-- 0xc8  (see "Default")

Volume Down:     BASE <-- 0x48  (volume down, no stereo detect,
                                 radio enable, tuner adjust disable)
                 wait 10 msec
                 BASE <-- 0xc8  (see "Default")

Volume Up:       BASE <-- 0x88  (volume up, no stereo detect,
                                 radio enable, tuner adjust disable)
                 wait 10 msec
                 BASE <-- 0xc8  (see "Default")

Check Stereo:    BASE <-- 0xd8  (current volume, stereo detect,
                                 radio enable, tuner adjust disable)
                 wait 100 msec
                 x <-- BASE     (read ioport)
                 BASE <-- 0xc8  (see "Default")

                 x=0xff ==> "not stereo", x=0xfd ==> "stereo detected"

Set Frequency:   code = (freq*40) + 10486188
                 foreach of the 24 bits in code,
                 (from Least to Most Significant):
                 to write a "zero" bit,
                 BASE <-- 0x01  (audio mute, no stereo detect, radio
                                 disable, "zero" bit phase 1, tuner adjust)
                 BASE <-- 0x03  (audio mute, no stereo detect, radio
                                 disable, "zero" bit phase 2, tuner adjust)
                 to write a "one" bit,
```

(continues on next page)

(continued from previous page)

```
                BASE <-- 0x05  (audio mute, no stereo detect, radio
                                disable, "one" bit phase 1, tuner adjust)
                BASE <-- 0x07  (audio mute, no stereo detect, radio
                                disable, "one" bit phase 2, tuner adjust)
```

The saa7134 driver

Author Gerd Hoffmann

Card Variations:

Cards can use either of these two crystals (xtal):

- 32.11 MHz -> .audio_clock=0x187de7
- 24.576MHz -> .audio_clock=0x200000 (xtal * .audio_clock = 51539600)

Some details about 30/34/35:

- saa7130 - low-price chip, doesn't have mute, that is why all those cards should have .mute field defined in their tuner structure.
- saa7134 - usual chip
- saa7133/35 - saa7135 is probably a marketing decision, since all those chips identifies itself as 33 on pci.

LifeView GPIOs

This section was authored by: Peter Missel <peter.missel@onlinehome.de>

- LifeView FlyTV Platinum FM (LR214WF)
  - GP27 MDT2005 PB4 pin 10
  - GP26 MDT2005 PB3 pin 9
  - GP25 MDT2005 PB2 pin 8
  - GP23 MDT2005 PB1 pin 7
  - GP22 MDT2005 PB0 pin 6
  - GP21 MDT2005 PB5 pin 11
  - GP20 MDT2005 PB6 pin 12
  - GP19 MDT2005 PB7 pin 13
  - nc MDT2005 PA3 pin 2
  - Remote MDT2005 PA2 pin 1
  - GP18 MDT2005 PA1 pin 18
  - nc MDT2005 PA0 pin 17 strap low

- GP17 Strap “GP7” =High
- GP16 Strap “GP6” =High
  - 0=Radio 1=TV
  - Drives SA630D ENCH1 and HEF4052 A1 pinsto do FM radio through SIF input
- GP15 nc
- GP14 nc
- GP13 nc
- GP12 Strap “GP5” = High
- GP11 Strap “GP4” = High
- GP10 Strap “GP3” = High
- GP09 Strap “GP2” = Low
- GP08 Strap “GP1” = Low
- GP07.00 nc

Credits

andrew.stevens@philips.com + werner.leeb@philips.com for providing saa7134 hardware specs and sample board.

Cropping and Scaling algorithm, used in the sh_mobile_ceu_camera driver

Author: Guennadi Liakhovetski <g.liakhovetski@gmx.de>

Terminology

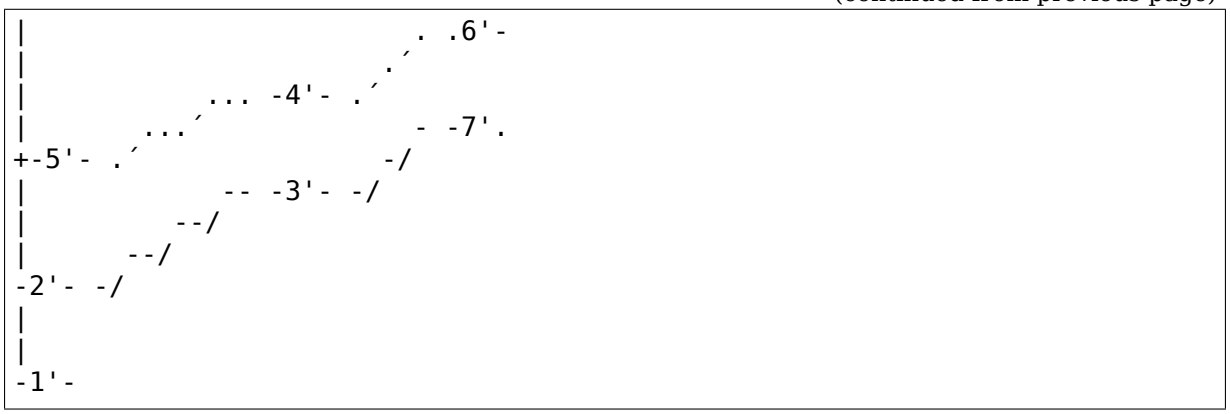
sensor scales: horizontal and vertical scales, configured by the sensor driver host scales: - “- host driver combined scales: sensor_scale * host_scale

Generic scaling / cropping scheme

```
-1--
|
-2-- -\
|       --\
|          --\
+-5-- .       -- -3-- -\
|       `...              -\
|           `... -4-- .    - -7..
|                      `.
|                        `. .6--
|
```

(continues on next page)

(continued from previous page)



In the above chart minuses and slashes represent “real” data amounts, points and accents represent “useful” data, basically, CEU scaled and cropped output, mapped back onto the client’s source plane.

Such a configuration can be produced by user requests:

S_CROP(left / top = (5) - (1), width / height = (5’) - (5)) S_FMT(width / height = (6’) - (6))

Here:

(1) to (1’) - whole max width or height (1) to (2) - sensor cropped left or top (2) to (2’) - sensor cropped width or height (3) to (3’) - sensor scale (3) to (4) - CEU cropped left or top (4) to (4’) - CEU cropped width or height (5) to (5’) - reverse sensor scale applied to CEU cropped width or height (2) to (5) - reverse sensor scale applied to CEU cropped left or top (6) to (6’) - CEU scale - user window

S_FMT

Do not touch input rectangle - it is already optimal.

1. Calculate current sensor scales:

   scale_s = ((2’) - (2)) / ((3’) - (3))

2. Calculate “effective” input crop (sensor subwindow) - CEU crop scaled back at current sensor scales onto input window - this is user S_CROP:

   width_u = (5’) - (5) = ((4’) - (4)) * scale_s

3. Calculate new combined scales from “effective” input window to requested user window:

   scale_comb = width_u / ((6’) - (6))

4. Calculate sensor output window by applying combined scales to real input window:

   width_s_out = ((7’) - (7)) = ((2’) - (2)) / scale_comb

5. Apply iterative sensor S_FMT for sensor output window.

   subdev->video_ops->s_fmt(.width = width_s_out)

6. Retrieve sensor output window (g_fmt)

7. Calculate new sensor scales:

   scale_s_new = ((3’)_new - (3)_new) / ((2’) - (2))

8. Calculate new CEU crop - apply sensor scales to previously calculated “effective” crop:

   width_ceu = (4’)_new - (4)_new = width_u / scale_s_new left_ceu = (4)_new - (3)_new = ((5) - (2)) / scale_s_new

9. Use CEU cropping to crop to the new window:

   ceu_crop(.width = width_ceu, .left = left_ceu)

10. Use CEU scaling to scale to the requested user window:

    scale_ceu = width_ceu / width

S_CROP

The V4L2 crop API says:

“···specification does not define an origin or units. However by convention drivers should horizontally count unscaled samples relative to 0H.”

We choose to follow the advise and interpret cropping units as client input pixels.

Cropping is performed in the following 6 steps:

1. Request exactly user rectangle from the sensor.
2. If smaller - iterate until a larger one is obtained. Result: sensor cropped to 2 : 2’, target crop 5 : 5’, current output format 6’ - 6.
3. In the previous step the sensor has tried to preserve its output frame as good as possible, but it could have changed. Retrieve it again.
4. Sensor scaled to 3 : 3’. Sensor’s scale is (2’ - 2) / (3’ - 3). Calculate intermediate window: 4’ - 4 = (5’ - 5) * (3’ - 3) / (2’ - 2)
5. Calculate and apply host scale = (6’ - 6) / (4’ - 4)
6. Calculate and apply host crop: 6 - 7 = (5 - 2) * (6’ - 6) / (5’ - 5)

Tuner drivers

Simple tuner Programming

There are some flavors of Tuner programming APIs. These differ mainly by the bandswitch byte.

- L= LG_API (VHF_LO=0x01, VHF_HI=0x02, UHF=0x08, radio=0x04)
- P= PHILIPS_API (VHF_LO=0xA0, VHF_HI=0x90, UHF=0x30, radio=0x04)
- T= TEMIC_API (VHF_LO=0x02, VHF_HI=0x04, UHF=0x01)
- A= ALPS_API (VHF_LO=0x14, VHF_HI=0x12, UHF=0x11)
- M= PHILIPS_MK3 (VHF_LO=0x01, VHF_HI=0x02, UHF=0x04, radio=0x19)

Tuner Manufacturers

- SAMSUNG Tuner identification: (e.g. TCPM9091PD27)

```
TCP [ABCJLMNQ] 90[89][125] [DP] [ACD] 27 [ABCD]
[ABCJLMNQ]:
  A= BG+DK
  B= BG
  C= I+DK
  J= NTSC-Japan
  L= Secam LL
  M= BG+I+DK
  N= NTSC
  Q= BG+I+DK+LL
[89]: ?
[125]:
  2: No FM
  5: With FM
[DP]:
  D= NTSC
  P= PAL
[ACD]:
  A= F-connector
  C= Phono connector
  D= Din Jack
[ABCD]:
  3-wire/I2C tuning, 2-band/3-band
```

These Tuners are PHILIPS_API compatible.

Philips Tuner identification: (e.g. FM1216MF)

```
F[IRMQ]12[1345]6{MF|ME|MP}
F[IRMQ]:
 FI12x6: Tuner Series
 FR12x6: Tuner + Radio IF
 FM12x6: Tuner + FM
 FQ12x6: special
 FMR12x6: special
 TD15xx: Digital Tuner ATSC
12[1345]6:
 1216: PAL BG
 1236: NTSC
 1246: PAL I
 1256: Pal DK
{MF|ME|MP}
 MF: BG LL w/ Secam (Multi France)
 ME: BG DK I LL   (Multi Europe)
 MP: BG DK I      (Multi PAL)
 MR: BG DK M (?)
 MG: BG DKI M (?)
MK2 series PHILIPS_API, most tuners are compatible to this one !
MK3 series introduced in 2002 w/ PHILIPS_MK3_API
```

Temic Tuner identification: (.e.g 4006FH5)

```
 4[01][0136][269]F[HYNR]5
  40x2: Tuner (5V/33V), TEMIC_API.
  40x6: Tuner 5V
  41xx: Tuner compact
  40x9: Tuner+FM compact
 [0136]
  xx0x: PAL BG
  xx1x: Pal DK, Secam LL
  xx3x: NTSC
  xx6x: PAL I
 F[HYNR]5
  FH5: Pal BG
  FY5: others
  FN5: multistandard
  FR5: w/ FM radio
 3X xxxx: order number with specific connector
Note: Only 40x2 series has TEMIC_API, all newer tuners have PHILIPS_API.
```

LG Innotek Tuner:

- TPI8NSR11 : NTSC J/M (TPI8NSR01 w/FM) (P,210/497)
- TPI8PSB11 : PAL B/G (TPI8PSB01 w/FM) (P,170/450)
- TAPC-I701 : PAL I (TAPC-I001 w/FM) (P,170/450)
- TPI8PSB12 : PAL D/K+B/G (TPI8PSB02 w/FM) (P,170/450)
- TAPC-H701P: NTSC_JP (TAPC-H001P w/FM) (L,170/450)
- TAPC-G701P: PAL B/G (TAPC-G001P w/FM) (L,170/450)
- TAPC-W701P: PAL I (TAPC-W001P w/FM) (L,170/450)
- TAPC-Q703P: PAL D/K (TAPC-Q001P w/FM) (L,170/450)
- TAPC-Q704P: PAL D/K+I (L,170/450)
- TAPC-G702P: PAL D/K+B/G (L,170/450)
- TADC-H002F: NTSC (L,175/410?; 2-B, C-W+11, W+12-69)
- TADC-M201D: PAL D/K+B/G+I (L,143/425) (sound control at I2C address 0xc8)
- TADC-T003F: NTSC Taiwan (L,175/410?; 2-B, C-W+11, W+12-69)

Suffix:

- P= Standard phono female socket
- D= IEC female socket
- F= F-connector

Other Tuners:

- TCL2002MB-1 : PAL BG + DK =TUNER_LG_PAL_NEW_TAPC
- TCL2002MB-1F: PAL BG + DK w/FM =PHILIPS_PAL
- TCL2002MI-2 : PAL I = ??

ALPS Tuners:

- Most are LG_API compatible
- TSCH6 has ALPS_API (TSCH5 ?)
- TSBE1 has extra API 05,02,08 Control_byte=0xCB Source:[1]

The Virtual Media Controller Driver (vimc)

Source code documentation

vimc-streamer

struct **vimc_stream**
struct that represents a stream in the pipeline

Definition

```
struct vimc_stream {
  struct media_pipeline pipe;
  struct vimc_ent_device *ved_pipeline[VIMC_STREAMER_PIPELINE_MAX_SIZE];
  unsigned int pipe_size;
  struct task_struct *kthread;
};
```

Members

pipe the media pipeline object associated with this stream

ved_pipeline array containing all the entities participating in the stream. The order is from a video device (usually a capture device) where stream_on was called, to the entity generating the first base image to be processed in the pipeline.

pipe_size size of **ved_pipeline**

kthread thread that generates the frames of the stream.

Description

When the user call stream_on in a video device, struct vimc_stream is used to keep track of all entities and subdevices that generates and process frames for the stream.

struct media_entity * **vimc_get_source_entity**(struct media_entity * ent)
get the entity connected with the first sink pad

Parameters

struct media_entity * ent reference media_entity

Description

Helper function that returns the media entity containing the source pad linked with the first sink pad from the given media entity pad list.

Return

The source pad or NULL, if it wasn’t found.

[1] conexant100029b-PCI-Decoder-ApplicationNote.pdf

void **vimc_streamer_pipeline_terminate**(struct vimc_stream * stream)
Disable stream in all ved in stream

Parameters

struct vimc_stream * stream the pointer to the stream structure with the pipeline to be disabled.

Description

Calls s_stream to disable the stream in each entity of the pipeline

int **vimc_streamer_pipeline_init**(struct vimc_stream * stream, struct vimc_ent_device * ved)
Initializes the stream structure

Parameters

struct vimc_stream * stream the pointer to the stream structure to be initialized

struct vimc_ent_device * ved the pointer to the vimc entity initializing the stream

Description

Initializes the stream structure. Walks through the entity graph to construct the pipeline used later on the streamer thread. Calls `vimc_streamer_s_stream()` to enable stream in all entities of the pipeline.

Return

0 if success, error code otherwise.

int **vimc_streamer_thread**(void * data)
Process frames through the pipeline

Parameters

void * data vimc_stream struct of the current stream

Description

From the source to the sink, gets a frame from each subdevice and send to the next one of the pipeline at a fixed framerate.

Return

Always zero (created as `int` instead of `void` to comply with kthread API).

int **vimc_streamer_s_stream**(struct vimc_stream * stream, struct vimc_ent_device * ved, int enable)
Start/stop the streaming on the media pipeline

Parameters

struct vimc_stream * stream the pointer to the stream structure of the current stream

struct vimc_ent_device * ved pointer to the vimc entity of the entity of the stream

int enable flag to determine if stream should start/stop

Description

When starting, check if there is no `stream->kthread` allocated. This should indicate that a stream is already running. Then, it initializes the pipeline, creates and runs a kthread to consume buffers through the pipeline. When stopping, analogously check if there is a stream running, stop the thread and terminates the pipeline.

Return

0 if success, error code otherwise.

53.7.2 Digital TV drivers

Idea behind the dvb-usb-framework

Note:

1) This documentation is outdated. Please check at the DVB wiki at https://linuxtv.org/wiki for more updated info.

2) **deprecated:** Newer DVB USB drivers should use the dvb-usb-v2 framework.

In March 2005 I got the new Twinhan USB2.0 DVB-T device. They provided specs and a firmware.

Quite keen I wanted to put the driver (with some quirks of course) into dibusb. After reading some specs and doing some USB snooping, it realized, that the dibusb-driver would be a complete mess afterwards. So I decided to do it in a different way: With the help of a dvb-usb-framework.

The framework provides generic functions (mostly kernel API calls), such as:

- Transport Stream URB handling in conjunction with dvb-demux-feed-control (bulk and isoc are supported)
- registering the device for the DVB-API
- registering an I2C-adapter if applicable
- remote-control/input-device handling
- firmware requesting and loading (currently just for the Cypress USB controllers)
- other functions/methods which can be shared by several drivers (such as functions for bulk-control-commands)
- TODO: a I2C-chunker. It creates device-specific chunks of register-accesses depending on length of a register and the number of values that can be multi-written and multi-read.

The source code of the particular DVB USB devices does just the communication with the device via the bus. The connection between the DVB-API-functionality is done via callbacks, assigned in a static device-description (struct dvb_usb_device) each device-driver has to have.

For an example have a look in drivers/media/usb/dvb-usb/vp7045*.

Objective is to migrate all the usb-devices (dibusb, cinergyT2, maybe the ttusb; flexcop-usb already benefits from the generic flexcop-device) to use the dvb-usb-lib.

TODO: dynamic enabling and disabling of the pid-filter in regard to number of feeds requested.

Supported devices

See the LinuxTV DVB Wiki at https://linuxtv.org for a complete list of cards/drivers/firmwares: https://linuxtv.org/wiki/index.php/DVB_USB

0. History & News:

   2005-06-30

   - added support for WideView WT-220U (Thanks to Steve Chang)

   2005-05-30

   - added basic isochronous support to the dvb-usb-framework
   - **added support for Conexant Hybrid reference design and Nebula** DigiTV USB

   2005-04-17

   - all dibusb devices ported to make use of the dvb-usb-framework

   2005-04-02

   - re-enabled and improved remote control code.

   2005-03-31

   - ported the Yakumo/Hama/Typhoon DVB-T USB2.0 device to dvb-usb.

   2005-03-30

   - first commit of the dvb-usb-module based on the dibusb-source. First device is a new driver for the TwinhanDTV Alpha / MagicBox II USB2.0-only DVB-T device.
   - (change from dvb-dibusb to dvb-usb)

   2005-03-28

   - added support for the AVerMedia AverTV DVB-T USB2.0 device (Thanks to Glen Harris and Jiun-Kuei Jung, AVerMedia)

   2005-03-14

   - added support for the Typhoon/Yakumo/HAMA DVB-T mobile USB2.0

   2005-02-11

   - added support for the KWorld/ADSTech Instant DVB-T USB2.0. Thanks a lot to Joachim von Caron

2005-02-02 - added support for the Hauppauge Win-TV Nova-T USB2

2005-01-31 - distorted streaming is gone for USB1.1 devices

2005-01-13

- moved the mirrored pid_filter_table back to dvb-dibusb first almost working version for HanfTek UMT-010 found out, that Yakumo/HAMA/Typhoon are predecessors of the HanfTek UMT-010

2005-01-10

- refactoring completed, now everything is very delightful
- tuner quirks for some weird devices (Artec T1 AN2235 device has sometimes a Panasonic Tuner assembled). Tunerprobing implemented. Thanks a lot to Gunnar Wittich.

2004-12-29

- after several days of struggling around bug of no returning URBs fixed.

2004-12-26

- refactored the dibusb-driver, split into separate files
- i2c-probing enabled

2004-12-06

- possibility for demod i2c-address probing
- new usb IDs (Compro, Artec)

2004-11-23

- merged changes from DiB3000MC_ver2.1
- revised the debugging
- possibility to deliver the complete TS for USB2.0

2004-11-21

- first working version of the dib3000mc/p frontend driver.

2004-11-12

- added additional remote control keys. Thanks to Uwe Hanke.

2004-11-07

- added remote control support. Thanks to David Matthews.

2004-11-05

- added support for a new devices (Grandtec/Avermedia/Artec)
- merged my changes (for dib3000mb/dibusb) to the FE_REFACTORING, because it became HEAD
- moved transfer control (pid filter, fifo control) from usb driver to frontend, it seems better settled there (added xfer_ops-struct)
- created a common files for frontends (mc/p/mb)

2004-09-28

- added support for a new device (Unknown, vendor ID is Hyper-Paltek)

2004-09-20

- added support for a new device (Compro DVB-U2000), thanks to Amaury Demol for reporting
- changed usb TS transfer method (several urbs, stopping transfer before setting a new pid)

2004-09-13

- added support for a new device (Artec T1 USB TVBOX), thanks to Christian Motschke for reporting

2004-09-05

- released the dibusb device and dib3000mb-frontend driver (old news for vp7041.c)

2004-07-15

- found out, by accident, that the device has a TUA6010XS for PLL

2004-07-12

- figured out, that the driver should also work with the CTS Portable (Chinese Television System)

2004-07-08

- firmware-extraction-2.422-problem solved, driver is now working properly with firmware extracted from 2.422
- #if for 2.6.4 (dvb), compile issue
- changed firmware handling, see vp7041.txt sec 1.1

2004-07-02

- some tuner modifications, v0.1, cleanups, first public

2004-06-28

- now using the dvb_dmx_swfilter_packets, everything runs fine now

2004-06-27

- able to watch and switching channels (pre-alpha)
- no section filtering yet

2004-06-06

- first TS received, but kernel oops :/

2004-05-14

- firmware loader is working

2004-05-11

- start writing the driver

How to use?

Firmware

Most of the USB drivers need to download a firmware to the device before start working.

Have a look at the Wikipage for the DVB-USB-drivers to find out, which firmware you need for your device:

https://linuxtv.org/wiki/index.php/DVB_USB

Compiling

Since the driver is in the linux kernel, activating the driver in your favorite config-environment should sufficient. I recommend to compile the driver as module. Hotplug does the rest.

If you use dvb-kernel enter the build-2.6 directory run ‘make’ and ‘insmod.sh load’ afterwards.

Loading the drivers

Hotplug is able to load the driver, when it is needed (because you plugged in the device).

If you want to enable debug output, you have to load the driver manually and from within the dvb-kernel cvs repository.

first have a look, which debug level are available:

```
# modinfo dvb-usb
# modinfo dvb-usb-vp7045

etc.
```

```
modprobe dvb-usb debug=<level>
modprobe dvb-usb-vp7045 debug=<level>
etc.
```

should do the trick.

When the driver is loaded successfully, the firmware file was in the right place and the device is connected, the “Power”-LED should be turned on.

At this point you should be able to start a dvb-capable application. I’m use (t|s)zap, mplayer and dvbscan to test the basics. VDR-xine provides the long-term test scenario.

Known problems and bugs

- Don’t remove the USB device while running an DVB application, your system will go crazy or die most likely.

Adding support for devices

TODO

USB1.1 Bandwidth limitation

A lot of the currently supported devices are USB1.1 and thus they have a maximum bandwidth of about 5-6 MBit/s when connected to a USB2.0 hub. This is not enough for receiving the complete transport stream of a DVB-T channel (which is about 16 MBit/s). Normally this is not a problem, if you only want to watch TV (this does not apply for HDTV), but watching a channel while recording another channel on the same frequency simply does not work very well. This applies to all USB1.1 DVB-T devices, not just the dvb-usb-devices)

The bug, where the TS is distorted by a heavy usage of the device is gone definitely. All dvb-usb-devices I was using (Twinhan, Kworld, DiBcom) are working like charm now with VDR. Sometimes I even was able to record a channel and watch another one.

Comments

Patches, comments and suggestions are very very welcome.

3. Acknowledgements

Amaury Demol (Amaury.Demol@parrot.com) and Francois Kanounnikoff from DiBcom for providing specs, code and help, on which the dvb-dibusb, dib3000mb and dib3000mc are based.

David Matthews for identifying a new device type (Artec T1 with AN2235) and for extending dibusb with remote control event handling. Thank you.

Alex Woods for frequently answering question about usb and dvb stuff, a big thank you.

Bernd Wagner for helping with huge bug reports and discussions.

Gunnar Wittich and Joachim von Caron for their trust for providing root-shells on their machines to implement support for new devices.

Allan Third and Michael Hutchinson for their help to write the Nebula digitv-driver.

Glen Harris for bringing up, that there is a new dibusb-device and Jiun-Kuei Jung from AVerMedia who kindly provided a special firmware to get the device up and running in Linux.

Jennifer Chen, Jeff and Jack from Twinhan for kindly supporting by writing the vp7045-driver.

Steve Chang from WideView for providing information for new devices and firmware files.

Michael Paxton for submitting remote control keymaps.

Some guys on the linux-dvb mailing list for encouraging me.

Peter Schildmann >peter.schildmann-nospam-at-web.de< for his user-level firmware loader, which saves a lot of time (when writing the vp7041 driver)

Ulf Hermenau for helping me out with traditional chinese.

André Smoktun and Christian Frömmel for supporting me with hardware and listening to my problems very patiently.

Frontend drivers

Frontend attach headers

struct **a8293_platform_data**
Platform data for the a8293 driver

Definition

```
struct a8293_platform_data {
  struct dvb_frontend *dvb_frontend;
};
```

Members

dvb_frontend DVB frontend.

struct **af9013_platform_data**
Platform data for the af9013 driver

Definition

```
struct af9013_platform_data {
  u32 clk;
#define AF9013_TUNER_MXL5003D      3 ;
#define AF9013_TUNER_MXL5005D     13 ;
#define AF9013_TUNER_MXL5005R     30 ;
#define AF9013_TUNER_ENV77H11D5  129 ;
#define AF9013_TUNER_MT2060      130 ;
#define AF9013_TUNER_MC44S803    133 ;
#define AF9013_TUNER_QT1010      134 ;
#define AF9013_TUNER_UNKNOWN     140 ;
#define AF9013_TUNER_MT2060_2    147 ;
#define AF9013_TUNER_TDA18271    156 ;
```

(continues on next page)

(continued from previous page)

```
#define AF9013_TUNER_QT1010A     162 ;
#define AF9013_TUNER_MXL5007T    177 ;
#define AF9013_TUNER_TDA18218    179 ;
  u8 tuner;
  u32 if_frequency;
#define AF9013_TS_MODE_USB       0;
#define AF9013_TS_MODE_PARALLEL  1;
#define AF9013_TS_MODE_SERIAL    2;
  u8 ts_mode;
  u8 ts_output_pin;
  bool spec_inv;
  u8 api_version[4];
#define AF9013_GPIO_ON (1 << 0);
#define AF9013_GPIO_EN (1 << 1);
#define AF9013_GPIO_O  (1 << 2);
#define AF9013_GPIO_I  (1 << 3);
#define AF9013_GPIO_LO (AF9013_GPIO_ON|AF9013_GPIO_EN);
#define AF9013_GPIO_HI (AF9013_GPIO_ON|AF9013_GPIO_EN|AF9013_GPIO_O);
#define AF9013_GPIO_TUNER_ON  (AF9013_GPIO_ON|AF9013_GPIO_EN);
#define AF9013_GPIO_TUNER_OFF (AF9013_GPIO_ON|AF9013_GPIO_EN|AF9013_GPIO_
↪O);
  u8 gpio[4];
  struct dvb_frontend* (*get_dvb_frontend)(struct i2c_client *);
  struct i2c_adapter* (*get_i2c_adapter)(struct i2c_client *);
  int (*pid_filter_ctrl)(struct dvb_frontend *, int);
  int (*pid_filter)(struct dvb_frontend *, u8, u16, int);
};
```

Members

clk Clock frequency.

tuner Used tuner model.

if_frequency IF frequency.

ts_mode TS mode.

ts_output_pin TS output pin.

spec_inv Input spectrum inverted.

api_version Firmware API version.

gpio GPIOs.

get_dvb_frontend Get DVB frontend callback.

get_i2c_adapter Get I2C adapter.

pid_filter_ctrl Control PID filter.

pid_filter Set PID to PID filter.

struct **ascot2e_config**
the configuration of Ascot2E tuner driver

Definition

```
struct ascot2e_config {
  u8 i2c_address;
  u8 xtal_freq_mhz;
  void *set_tuner_priv;
  int (*set_tuner_callback)(void *, int);
};
```

Members

i2c_address I2C address of the tuner

xtal_freq_mhz Oscillator frequency, MHz

set_tuner_priv Callback function private context

set_tuner_callback Callback function that notifies the parent driver which tuner is active now

struct dvb_frontend * **ascot2e_attach**(struct dvb_frontend * fe, const struct ascot2e_config * config, struct i2c_adapter * i2c)

Parameters

struct dvb_frontend * fe frontend to be attached

const struct ascot2e_config * config pointer to `struct ascot2e_config` with tuner configuration.

struct i2c_adapter * i2c i2c adapter to use.

Return

FE pointer on success, NULL on failure.

struct **cxd2820r_platform_data**
Platform data for the cxd2820r driver

Definition

```
struct cxd2820r_platform_data {
  u8 ts_mode;
  bool ts_clk_inv;
  bool if_agc_polarity;
  bool spec_inv;
  int **gpio_chip_base;
  struct dvb_frontend* (*get_dvb_frontend)(struct i2c_client *);
};
```

Members

ts_mode TS mode.

ts_clk_inv TS clock inverted.

if_agc_polarity IF AGC polarity.

spec_inv Input spectrum inverted.

gpio_chip_base GPIO.

get_dvb_frontend Get DVB frontend.

struct **cxd2820r_config**
configuration for cxd2020r demod

Definition

```
struct cxd2820r_config {
  u8 i2c_address;
  u8 ts_mode;
  bool ts_clock_inv;
  bool if_agc_polarity;
  bool spec_inv;
};
```

Members

i2c_address Demodulator I2C address. Driver determines DVB-C slave I2C address automatically from master address. Default: none, must set. Values: 0x6c, 0x6d.

ts_mode TS output mode. Default: none, must set. Values: FIXME?

ts_clock_inv TS clock inverted. Default: 0. Values: 0, 1.

if_agc_polarity Default: 0. Values: 0, 1

spec_inv Spectrum inversion. Default: 0. Values: 0, 1.

struct dvb_frontend * **cxd2820r_attach**(const struct cxd2820r_config * config, struct i2c_adapter * i2c, int * gpio_chip_base)

Parameters

const struct cxd2820r_config * config pointer to `struct cxd2820r_config` with demod configuration.

struct i2c_adapter * i2c i2c adapter to use.

int * gpio_chip_base if zero, disables GPIO setting. Otherwise, if CONFIG_GPIOLIB is set dynamically allocate gpio base; if is not set, use its value to setup the GPIO pins.

Return

FE pointer on success, NULL on failure.

struct **drxk_config**
Configure the initial parameters for DRX-K

Definition

```
struct drxk_config {
  u8 adr;
  bool single_master;
  bool no_i2c_bridge;
  bool parallel_ts;
  bool dynamic_clk;
  bool enable_merr_cfg;
  bool antenna_dvbt;
  u16 antenna_gpio;
```

(continues on next page)

(continued from previous page)

```
  u8 mpeg_out_clk_strength;
  int chunk_size;
  const char      *microcode_name;
  int qam_demod_parameter_count;
};
```

Members

adr I2C address of the DRX-K

single_master Device is on the single master mode

no_i2c_bridge Don't switch the I2C bridge to talk with tuner

parallel_ts True means that the device uses parallel TS, Serial otherwise.

dynamic_clk True means that the clock will be dynamically adjusted. Static clock otherwise.

enable_merr_cfg Enable SIO_PDR_PERR_CFG/SIO_PDR_MVAL_CFG.

antenna_dvbt GPIO bit for changing antenna to DVB-C. A value of 1 means that 1=DVBC, 0 = DVBT. Zero means the opposite.

antenna_gpio GPIO bit used to control the antenna

mpeg_out_clk_strength DRXK Mpeg output clock drive strength.

chunk_size maximum size for I2C messages

microcode_name Name of the firmware file with the microcode

qam_demod_parameter_count The number of parameters used for the command to set the demodulator parameters. All firmwares are using the 2-parameter command. An exception is the `drxk_a3.mc` firmware, which uses the 4-parameter command. A value of 0 (default) or lower indicates that the correct number of parameters will be automatically detected.

Description

On the *`_gpio` vars, bit 0 is UIO-1, bit 1 is UIO-2 and bit 2 is UIO-3.

struct dvb_frontend * **drxk_attach**(const struct drxk_config * config, struct i2c_adapter * i2c)

Parameters

const struct drxk_config * config pointer to `struct drxk_config` with demod configuration.

struct i2c_adapter * i2c i2c adapter to use.

Return

FE pointer on success, NULL on failure.

struct dvb_frontend * **dvb_pll_attach**(struct dvb_frontend * fe, int pll_addr, struct i2c_adapter * i2c, unsigned int pll_desc_id)

pll to the supplied frontend structure.

Parameters

struct dvb_frontend * fe Frontend to attach to.

int pll_addr i2c address of the PLL (if used).

struct i2c_adapter * i2c i2c adapter to use (set to NULL if not used).

unsigned int pll_desc_id dvb_pll_desc to use.

Return

Frontend pointer on success, NULL on failure

struct **helene_config**
the configuration of 'Helene' tuner driver

Definition

```
struct helene_config {
  u8 i2c_address;
  u8 xtal_freq_mhz;
  void *set_tuner_priv;
  int (*set_tuner_callback)(void *, int);
  enum helene_xtal xtal;
  struct dvb_frontend *fe;
};
```

Members

i2c_address I2C address of the tuner

xtal_freq_mhz Oscillator frequency, MHz

set_tuner_priv Callback function private context

set_tuner_callback Callback function that notifies the parent driver which tuner is active now

xtal Cristal frequency as described by `enum helene_xtal`

fe Frontend for which connects this tuner

struct dvb_frontend * **helene_attach**(struct dvb_frontend * fe, const struct helene_config * config, struct i2c_adapter * i2c)

Parameters

struct dvb_frontend * fe frontend to be attached

const struct helene_config * config pointer to `struct helene_config` with tuner configuration.

struct i2c_adapter * i2c i2c adapter to use.

Return

FE pointer on success, NULL on failure.

struct dvb_frontend * **helene_attach_s**(struct dvb_frontend * fe, const struct helene_config * config, struct i2c_adapter * i2c)

Parameters

struct dvb_frontend * fe frontend to be attached

const struct helene_config * config pointer to `struct helene_config` with tuner configuration.

struct i2c_adapter * i2c i2c adapter to use.

Return

FE pointer on success, NULL on failure.

struct **horus3a_config**
the configuration of Horus3A tuner driver

Definition

```
struct horus3a_config {
  u8 i2c_address;
  u8 xtal_freq_mhz;
  void *set_tuner_priv;
  int (*set_tuner_callback)(void *, int);
};
```

Members

i2c_address I2C address of the tuner

xtal_freq_mhz Oscillator frequency, MHz

set_tuner_priv Callback function private context

set_tuner_callback Callback function that notifies the parent driver which tuner is active now

struct dvb_frontend * **horus3a_attach**(struct dvb_frontend * fe, const struct horus3a_config * config, struct i2c_adapter * i2c)

Parameters

struct dvb_frontend * fe frontend to be attached

const struct horus3a_config * config pointer to `struct helene_config` with tuner configuration.

struct i2c_adapter * i2c i2c adapter to use.

Return

FE pointer on success, NULL on failure.

DVB_IX2505V_H()
S silicon tuner

Parameters

Description

Copyright (C) 2010 Malcolm Priestley

struct **ix2505v_config**
ix2505 attachment configuration

Definition

```
struct ix2505v_config {
  u8 tuner_address;
  u8 tuner_gain;
  u8 tuner_chargepump;
  int min_delay_ms;
  u8 tuner_write_only;
};
```

Members

tuner_address tuner address

tuner_gain Baseband AMP gain control 0/1=0dB(default) 2=-2bB 3=-4dB

tuner_chargepump Charge pump output +/- 0=120 1=260 2=555 3=1200(default)

min_delay_ms delay after tune

tuner_write_only disables reads

struct dvb_frontend * **ix2505v_attach**(struct dvb_frontend * fe, const struct ix2505v_config * config, struct i2c_adapter * i2c)

Parameters

struct dvb_frontend * fe Frontend to attach to.

const struct ix2505v_config * config pointer to `struct ix2505v_config`

struct i2c_adapter * i2c pointer to `struct i2c_adapter`.

Return

FE pointer on success, NULL on failure.

enum **m88ds3103_ts_mode**
TS connection mode

Constants

M88DS3103_TS_SERIAL TS output pin D0, normal

M88DS3103_TS_SERIAL_D7 TS output pin D7

M88DS3103_TS_PARALLEL TS Parallel mode

M88DS3103_TS_CI TS CI Mode

enum **m88ds3103_clock_out**

Constants

M88DS3103_CLOCK_OUT_DISABLED Clock output is disabled

M88DS3103_CLOCK_OUT_ENABLED Clock output is enabled with crystal clock.

M88DS3103_CLOCK_OUT_ENABLED_DIV2 Clock output is enabled with half crystal clock.

struct **m88ds3103_platform_data**
Platform data for the m88ds3103 driver

Definition

```
struct m88ds3103_platform_data {
  u32 clk;
  u16 i2c_wr_max;
  enum m88ds3103_ts_mode ts_mode;
  u32 ts_clk;
  enum m88ds3103_clock_out clk_out;
  u8 ts_clk_pol:1;
  u8 spec_inv:1;
  u8 agc;
  u8 agc_inv:1;
  u8 envelope_mode:1;
  u8 lnb_hv_pol:1;
  u8 lnb_en_pol:1;
  struct dvb_frontend* (*get_dvb_frontend)(struct i2c_client *);
  struct i2c_adapter* (*get_i2c_adapter)(struct i2c_client *);
};
```

Members

clk Clock frequency.

i2c_wr_max Max bytes I2C adapter can write at once.

ts_mode TS mode.

ts_clk TS clock (KHz).

clk_out Clock output.

ts_clk_pol TS clk polarity. 1-active at falling edge; 0-active at rising edge.

spec_inv Input spectrum inversion.

agc AGC configuration.

agc_inv AGC polarity.

envelope_mode DiSEqC envelope mode.

lnb_hv_pol LNB H/V pin polarity. 0: pin high set to VOLTAGE_18, pin low to set VOLTAGE_13. 1: pin high set to VOLTAGE_13, pin low to set VOLTAGE_18.

lnb_en_pol LNB enable pin polarity. 0: pin high to disable, pin low to enable. 1: pin high to enable, pin low to disable.

get_dvb_frontend Get DVB frontend.

get_i2c_adapter Get I2C adapter.

struct **m88ds3103_config**
m88ds3102 configuration

Definition

```
struct m88ds3103_config {
  u8 i2c_addr;
  u32 clock;
  u16 i2c_wr_max;
  u8 ts_mode;
```

(continues on next page)

(continued from previous page)

```
  u32 ts_clk;
  u8 ts_clk_pol:1;
  u8 spec_inv:1;
  u8 agc_inv:1;
  u8 clock_out;
  u8 envelope_mode:1;
  u8 agc;
  u8 lnb_hv_pol:1;
  u8 lnb_en_pol:1;
};
```

Members

i2c_addr I2C address. Default: none, must set. Example: 0x68, ⋯

clock Device' s clock. Default: none, must set. Example: 27000000

i2c_wr_max Max bytes I2C provider is asked to write at once. Default: none, must set. Example: 33, 65, ⋯

ts_mode TS output mode, as defined by `enum m88ds3103_ts_mode`. Default: M88DS3103_TS_SERIAL.

ts_clk TS clk in KHz. Default: 0.

ts_clk_pol TS clk polarity.Default: 0. 1-active at falling edge; 0-active at rising edge.

spec_inv Spectrum inversion. Default: 0.

agc_inv AGC polarity. Default: 0.

clock_out Clock output, as defined by `enum m88ds3103_clock_out`. Default: M88DS3103_CLOCK_OUT_DISABLED.

envelope_mode DiSEqC envelope mode. Default: 0.

agc AGC configuration. Default: none, must set.

lnb_hv_pol LNB H/V pin polarity. Default: 0. Values: 1: pin high set to VOLTAGE_13, pin low to set VOLTAGE_18; 0: pin high set to VOLTAGE_18, pin low to set VOLTAGE_13.

lnb_en_pol LNB enable pin polarity. Default: 0. Values: 1: pin high to enable, pin low to disable; 0: pin high to disable, pin low to enable.

struct dvb_frontend * **m88ds3103_attach**(const struct m88ds3103_config * config, struct i2c_adapter * i2c, struct i2c_adapter ** tuner_i2c)

Parameters

const struct m88ds3103_config * config pointer to `struct m88ds3103_config` with demod configuration.

struct i2c_adapter * i2c i2c adapter to use.

struct i2c_adapter ** tuner_i2c on success, returns the I2C adapter associated with m88ds3103 tuner.

Return

FE pointer on success, NULL on failure.

Note

Do not add new `m88ds3103_attach()` users! Use I2C bindings instead.

struct **mb86a20s_config**
Define the per-device attributes of the frontend

Definition

```
struct mb86a20s_config {
  u32 fclk;
  u8 demod_address;
  bool is_serial;
};
```

Members

fclk Clock frequency. If zero, assumes the default (32.57142 Mhz)

demod_address the demodulator's i2c address

is_serial if true, TS is serial. Otherwise, TS is parallel

struct dvb_frontend * **mb86a20s_attach**(const struct mb86a20s_config * config, struct i2c_adapter * i2c)

Parameters

const struct mb86a20s_config * config pointer to `struct mb86a20s_config` with demod configuration.

struct i2c_adapter * i2c i2c adapter to use.

Return

FE pointer on success, NULL on failure.

struct **mn88472_config**
Platform data for the mn88472 driver

Definition

```
struct mn88472_config {
  unsigned int xtal;
#define MN88472_TS_MODE_SERIAL      0;
#define MN88472_TS_MODE_PARALLEL    1;
  int ts_mode;
#define MN88472_TS_CLK_FIXED        0;
#define MN88472_TS_CLK_VARIABLE     1;
  int ts_clock;
  u16 i2c_wr_max;
  struct dvb_frontend **fe;
  struct dvb_frontend* (*get_dvb_frontend)(struct i2c_client *);
};
```

Members

xtal Clock frequency.

ts_mode TS mode.

ts_clock TS clock config.

i2c_wr_max Max number of bytes driver writes to I2C at once.

fe pointer to a frontend pointer

get_dvb_frontend Get DVB frontend callback.

struct **rtl2830_platform_data**
Platform data for the rtl2830 driver

Definition

```
struct rtl2830_platform_data {
  u32 clk;
  bool spec_inv;
  u8 vtop;
  u8 krf;
  u8 agc_targ_val;
  struct dvb_frontend* (*get_dvb_frontend)(struct i2c_client *);
  struct i2c_adapter* (*get_i2c_adapter)(struct i2c_client *);
  int (*pid_filter)(struct dvb_frontend *, u8, u16, int);
  int (*pid_filter_ctrl)(struct dvb_frontend *, int);
};
```

Members

clk Clock frequency (4000000, 16000000, 25000000, 28800000).

spec_inv Spectrum inversion.

vtop AGC take-over point.

krf AGC ratio.

agc_targ_val AGC.

get_dvb_frontend Get DVB frontend.

get_i2c_adapter Get I2C adapter.

pid_filter Set PID to PID filter.

pid_filter_ctrl Control PID filter.

struct **rtl2832_platform_data**
Platform data for the rtl2832 driver

Definition

```
struct rtl2832_platform_data {
  u32 clk;
#define RTL2832_TUNER_FC2580    0x21;
#define RTL2832_TUNER_TUA9001   0x24;
#define RTL2832_TUNER_FC0012    0x26;
#define RTL2832_TUNER_E4000     0x27;
#define RTL2832_TUNER_FC0013    0x29;
#define RTL2832_TUNER_R820T     0x2a;
#define RTL2832_TUNER_R828D     0x2b;
#define RTL2832_TUNER_SI2157    0x2c;
```

(continues on next page)

(continued from previous page)

```
  u8 tuner;
  struct dvb_frontend* (*get_dvb_frontend)(struct i2c_client *);
  struct i2c_adapter* (*get_i2c_adapter)(struct i2c_client *);
  int (*slave_ts_ctrl)(struct i2c_client *, bool);
  int (*pid_filter)(struct dvb_frontend *, u8, u16, int);
  int (*pid_filter_ctrl)(struct dvb_frontend *, int);
};
```

Members

clk Clock frequency (4000000, 16000000, 25000000, 28800000).

tuner Used tuner model.

get_dvb_frontend Get DVB frontend.

get_i2c_adapter Get I2C adapter.

slave_ts_ctrl Control slave TS interface.

pid_filter Set PID to PID filter.

pid_filter_ctrl Control PID filter.

struct **rtl2832_sdr_platform_data**
    Platform data for the rtl2832_sdr driver

Definition

```
struct rtl2832_sdr_platform_data {
  u32 clk;
#define RTL2832_SDR_TUNER_FC2580    0x21;
#define RTL2832_SDR_TUNER_TUA9001   0x24;
#define RTL2832_SDR_TUNER_FC0012    0x26;
#define RTL2832_SDR_TUNER_E4000     0x27;
#define RTL2832_SDR_TUNER_FC0013    0x29;
#define RTL2832_SDR_TUNER_R820T     0x2a;
#define RTL2832_SDR_TUNER_R828D     0x2b;
  u8 tuner;
  struct regmap *regmap;
  struct dvb_frontend *dvb_frontend;
  struct v4l2_subdev *v4l2_subdev;
  struct dvb_usb_device *dvb_usb_device;
};
```

Members

clk Clock frequency (4000000, 16000000, 25000000, 28800000).

tuner Used tuner model.

regmap pointer to `struct regmap`.

dvb_frontend rtl2832 DVB frontend.

v4l2_subdev Tuner v4l2 controls.

dvb_usb_device DVB USB interface for USB streaming.

struct dvb_frontend * **stb6000_attach**(struct dvb_frontend * fe, int addr,
    struct i2c_adapter * i2c)

Parameters

struct dvb_frontend * fe Frontend to attach to.

int addr i2c address of the tuner.

struct i2c_adapter * i2c i2c adapter to use.

Return

FE pointer on success, NULL on failure.

struct **tda10071_platform_data**
Platform data for the tda10071 driver

Definition

```
struct tda10071_platform_data {
  u32 clk;
  u16 i2c_wr_max;
#define TDA10071_TS_SERIAL        0;
#define TDA10071_TS_PARALLEL      1;
  u8 ts_mode;
  bool spec_inv;
  u8 pll_multiplier;
  u8 tuner_i2c_addr;
  struct dvb_frontend* (*get_dvb_frontend)(struct i2c_client *);
};
```

Members

clk Clock frequency.

i2c_wr_max Max bytes I2C adapter can write at once.

ts_mode TS mode.

spec_inv Input spectrum inversion.

pll_multiplier PLL multiplier.

tuner_i2c_addr CX24118A tuner I2C address (0x14, 0x54, ⋯).

get_dvb_frontend Get DVB frontend.

struct dvb_frontend* **tda826x_attach**(struct dvb_frontend * fe, int addr, struct i2c_adapter * i2c, int has_loopthrough)

Parameters

struct dvb_frontend * fe Frontend to attach to.

int addr i2c address of the tuner.

struct i2c_adapter * i2c i2c adapter to use.

int has_loopthrough Set to 1 if the card has a loopthrough RF connector.

Return

FE pointer on success, NULL on failure.

struct **zd1301_demod_platform_data**
Platform data for the zd1301_demod driver

Definition

```
struct zd1301_demod_platform_data {
  void *reg_priv;
  int (*reg_read)(void *, u16, u8 *);
  int (*reg_write)(void *, u16, u8);
};
```

Members

reg_priv First argument of reg_read and reg_write callbacks.

reg_read Register read callback.

reg_write Register write callback.

struct dvb_frontend * **zd1301_demod_get_dvb_frontend**(struct platform_device * pdev)
Get pointer to DVB frontend

Parameters

struct platform_device * pdev Pointer to platform device

Return

Pointer to DVB frontend which given platform device owns.

struct i2c_adapter * **zd1301_demod_get_i2c_adapter**(struct platform_device * pdev)
Get pointer to I2C adapter

Parameters

struct platform_device * pdev Pointer to platform device

Return

Pointer to I2C adapter which given platform device owns.

struct dvb_frontend * **zd1301_demod_get_dvb_frontend**(struct platform_device * dev)
Attach a zd1301 frontend

Parameters

struct platform_device * dev Pointer to platform device

Return

Pointer to `struct` dvb_frontend or NULL if attach fails.

DVB_ZL10036_H()
S silicon tuner

Parameters

Description

Copyright (C) 2006 Tino Reichardt Copyright (C) 2007-2009 Matthias Schwarzott <zzam**gentoo.de**>

struct dvb_frontend * **zl10036_attach**(struct dvb_frontend * fe, const struct zl10036_config * config, struct i2c_adapter * i2c)

Parameters

struct dvb_frontend * fe Frontend to attach to.

const struct zl10036_config * config zl10036_config structure.

struct i2c_adapter * i2c pointer to struct i2c_adapter.

Return

FE pointer on success, NULL on failure.

Contributors

Note: This documentation is outdated. There are several other DVB contributors that aren't listed below.

Thanks go to the following people for patches and contributions:

- Michael Hunold <m.hunold@gmx.de>
  - for the initial saa7146 driver and its recent overhaul
- Christian Theiss
  - for his work on the initial Linux DVB driver
- Marcus Metzler <mocm@metzlerbros.de> and Ralph Metzler <rjkm@metzlerbros.de>
  - for their continuing work on the DVB driver
- Michael Holzt <kju@debian.org>
  - for his contributions to the dvb-net driver
- Diego Picciani <d.picciani@novacomp.it>
  - for CyberLogin for Linux which allows logging onto EON (in case you are wondering where CyberLogin is, EON changed its login procedure and CyberLogin is no longer used.)
- Martin Schaller <martin@smurf.franken.de>
  - for patching the cable card decoder driver
- Klaus Schmidinger <Klaus.Schmidinger@cadsoft.de>
  - for various fixes regarding tuning, OSD and CI stuff and his work on VDR
- Steve Brown <sbrown@cortland.com>
  - for his AFC kernel thread

- Christoph Martin <martin@uni-mainz.de>
  - for his LIRC infrared handler
- Andreas Oberritter <obi@linuxtv.org>, Dennis Noermann <dennis.noermann@noernet.de>, Felix Domke <tmbinc@elitedvb.net>, Florian Schirmer <jolt@tuxbox.org>, Ronny Strutz <3des@elitedvb.de>, Wolfram Joost <dbox2@frokaschwei.de> and all the other dbox2 people
  - for many bugfixes in the generic DVB Core, frontend drivers and their work on the dbox2 port of the DVB driver
- Oliver Endriss <o.endriss@gmx.de>
  - for many bugfixes
- Andrew de Quincey <adq_dvb@lidskialf.net>
  - for the tda1004x frontend driver, and various bugfixes
- Peter Schildmann <peter.schildmann@web.de>
  - for the driver for the Technisat SkyStar2 PCI DVB card
- Vadim Catana <skystar@moldova.cc>, Roberto Ragusa <r.ragusa@libero.it> and Augusto Cardoso <augusto@carhil.net>
  - for all the work for the FlexCopII chipset by B2C2,Inc.
- Davor Emard <emard@softhome.net>
  - for his work on the budget drivers, the demux code, the module unloading problems, ⋯
- Hans-Frieder Vogt <hfvogt@arcor.de>
  - for his work on calculating and checking the crc’ s for the TechnoTrend/Hauppauge DEC driver firmware
- Michael Dreher <michael@5dot1.de> and Andreas ‘randy’ Weinberger
  - for the support of the Fujitsu-Siemens Activy budget DVB-S
- Kenneth Aafløy <ke-aa@frisurf.no>
  - for adding support for Typhoon DVB-S budget card
- Ernst Peinlich <e.peinlich@inode.at>
  - for tuning/DiSEqC support for the DEC 3000-s
- Peter Beutner <p.beutner@gmx.net>
  - for the IR code for the ttusb-dec driver
- Wilson Michaels <wilsonmichaels@earthlink.net>
  - for the lgdt330x frontend driver, and various bugfixes
- Michael Krufky <mkrufky@linuxtv.org>
  - for maintaining v4l/dvb inter-tree dependencies
- Taylor Jacob <rtjacob@earthlink.net>
  - for the nxt2002 frontend driver

- Jean-Francois Thibert <jeanfrancois@sagetv.com>
  - for the nxt2004 frontend driver
- Kirk Lapray <kirk.lapray@gmail.com>
  - for the or51211 and or51132 frontend drivers, and for merging the nxt2002 and nxt2004 modules into a single nxt200x frontend driver.

(If you think you should be in this list, but you are not, drop a line to the DVB mailing list)

CHAPTER

FIFTYFOUR

MISCELLANEOUS DEVICES

int **misc_register**(struct miscdevice * misc)
register a miscellaneous device

Parameters

struct miscdevice * misc device structure

Register a miscellaneous device with the kernel. If the minor number is set to `MISC_DYNAMIC_MINOR` a minor number is assigned and placed in the minor field of the structure. For other cases the minor number requested is used.

The structure passed is linked into the kernel and may not be destroyed until it has been unregistered. By default, an open() syscall to the device sets file->private_data to point to the structure. Drivers don't need open in fops for this.

A zero is returned on success and a negative errno code for failure.

void **misc_deregister**(struct miscdevice * misc)
unregister a miscellaneous device

Parameters

struct miscdevice * misc device to unregister

Unregister a miscellaneous device that was previously successfully registered with `misc_register()`.

CHAPTER

FIFTYFIVE

NEAR FIELD COMMUNICATION

55.1 HCI backend for NFC Core

- Author: Eric Lapuyade, Samuel Ortiz
- Contact: eric.lapuyade@intel.com, samuel.ortiz@intel.com

55.1.1 General

The HCI layer implements much of the ETSI TS 102 622 V10.2.0 specification. It enables easy writing of HCI-based NFC drivers. The HCI layer runs as an NFC Core backend, implementing an abstract nfc device and translating NFC Core API to HCI commands and events.

55.1.2 HCI

HCI registers as an nfc device with NFC Core. Requests coming from userspace are routed through netlink sockets to NFC Core and then to HCI. From this point, they are translated in a sequence of HCI commands sent to the HCI layer in the host controller (the chip). Commands can be executed synchronously (the sending context blocks waiting for response) or asynchronously (the response is returned from HCI Rx context). HCI events can also be received from the host controller. They will be handled and a translation will be forwarded to NFC Core as needed. There are hooks to let the HCI driver handle proprietary events or override standard behavior. HCI uses 2 execution contexts:

- one for executing commands : nfc_hci_msg_tx_work(). Only one command can be executing at any given moment.
- one for dispatching received events and commands : nfc_hci_msg_rx_work().

55.1.3 HCI Session initialization

The Session initialization is an HCI standard which must unfortunately support proprietary gates. This is the reason why the driver will pass a list of proprietary gates that must be part of the session. HCI will ensure all those gates have pipes connected when the hci device is set up. In case the chip supports pre-opened gates and pseudo-static pipes, the driver can pass that information to HCI core.

55.1.4 HCI Gates and Pipes

A gate defines the 'port' where some service can be found. In order to access a service, one must create a pipe to that gate and open it. In this implementation, pipes are totally hidden. The public API only knows gates. This is consistent with the driver need to send commands to proprietary gates without knowing the pipe connected to it.

55.1.5 Driver interface

A driver is generally written in two parts : the physical link management and the HCI management. This makes it easier to maintain a driver for a chip that can be connected using various phy (i2c, spi, ⋯)

55.1.6 HCI Management

A driver would normally register itself with HCI and provide the following entry points:

```
struct nfc_hci_ops {
      int (*open)(struct nfc_hci_dev *hdev);
      void (*close)(struct nfc_hci_dev *hdev);
      int (*hci_ready) (struct nfc_hci_dev *hdev);
      int (*xmit) (struct nfc_hci_dev *hdev, struct sk_buff *skb);
      int (*start_poll) (struct nfc_hci_dev *hdev,
                         u32 im_protocols, u32 tm_protocols);
      int (*dep_link_up)(struct nfc_hci_dev *hdev, struct nfc_target␣
↪*target,
                         u8 comm_mode, u8 *gb, size_t gb_len);
      int (*dep_link_down)(struct nfc_hci_dev *hdev);
      int (*target_from_gate) (struct nfc_hci_dev *hdev, u8 gate,
                               struct nfc_target *target);
      int (*complete_target_discovered) (struct nfc_hci_dev *hdev, u8 gate,
                                         struct nfc_target *target);
      int (*im_transceive) (struct nfc_hci_dev *hdev,
                            struct nfc_target *target, struct sk_buff *skb,
                            data_exchange_cb_t cb, void *cb_context);
      int (*tm_send)(struct nfc_hci_dev *hdev, struct sk_buff *skb);
      int (*check_presence)(struct nfc_hci_dev *hdev,
                            struct nfc_target *target);
      int (*event_received)(struct nfc_hci_dev *hdev, u8 gate, u8 event,
                            struct sk_buff *skb);
};
```

- open() and close() shall turn the hardware on and off.
- hci_ready() is an optional entry point that is called right after the hci session has been set up. The driver can use it to do additional initialization that must be performed using HCI commands.
- xmit() shall simply write a frame to the physical link.
- start_poll() is an optional entrypoint that shall set the hardware in polling mode. This must be implemented only if the hardware uses proprietary gates or a mechanism slightly different from the HCI standard.
- dep_link_up() is called after a p2p target has been detected, to finish the p2p connection setup with hardware parameters that need to be passed back to nfc core.
- dep_link_down() is called to bring the p2p link down.
- target_from_gate() is an optional entrypoint to return the nfc protocols corresponding to a proprietary gate.
- complete_target_discovered() is an optional entry point to let the driver perform additional proprietary processing necessary to auto activate the discovered target.
- im_transceive() must be implemented by the driver if proprietary HCI commands are required to send data to the tag. Some tag types will require custom commands, others can be written to using the standard HCI commands. The driver can check the tag type and either do proprietary processing, or return 1 to ask for standard processing. The data exchange command itself must be sent asynchronously.
- tm_send() is called to send data in the case of a p2p connection
- check_presence() is an optional entry point that will be called regularly by the core to check that an activated tag is still in the field. If this is not implemented, the core will not be able to push tag_lost events to the user space
- event_received() is called to handle an event coming from the chip. Driver can handle the event or return 1 to let HCI attempt standard processing.

On the rx path, the driver is responsible to push incoming HCP frames to HCI using nfc_hci_recv_frame(). HCI will take care of re-aggregation and handling This must be done from a context that can sleep.

55.1.7 PHY Management

The physical link (i2c, ⋯) management is defined by the following structure:

```
struct nfc_phy_ops {
      int (*write)(void *dev_id, struct sk_buff *skb);
      int (*enable)(void *dev_id);
      void (*disable)(void *dev_id);
};
```

enable(): turn the phy on (power on), make it ready to transfer data

disable(): turn the phy off

write(): Send a data frame to the chip. Note that to enable higher layers such as an llc to store the frame for re-emission, this function must not alter the skb. It must also not return a positive result (return 0 for success, negative for failure).

Data coming from the chip shall be sent directly to nfc_hci_recv_frame().

55.1.8 LLC

Communication between the CPU and the chip often requires some link layer protocol. Those are isolated as modules managed by the HCI layer. There are currently two modules : nop (raw transfert) and shdlc. A new llc must implement the following functions:

```
struct nfc_llc_ops {
      void *(*init) (struct nfc_hci_dev *hdev, xmit_to_drv_t xmit_to_drv,
                     rcv_to_hci_t rcv_to_hci, int tx_headroom,
                     int tx_tailroom, int *rx_headroom, int *rx_tailroom,
                     llc_failure_t llc_failure);
      void (*deinit) (struct nfc_llc *llc);
      int (*start) (struct nfc_llc *llc);
      int (*stop) (struct nfc_llc *llc);
      void (*rcv_from_drv) (struct nfc_llc *llc, struct sk_buff *skb);
      int (*xmit_from_hci) (struct nfc_llc *llc, struct sk_buff *skb);
};
```

init(): allocate and init your private storage

deinit(): cleanup

start(): establish the logical connection

stop (): terminate the logical connection

rcv_from_drv(): handle data coming from the chip, going to HCI

xmit_from_hci(): handle data sent by HCI, going to the chip

The llc must be registered with nfc before it can be used. Do that by calling:

```
nfc_llc_register(const char *name, struct nfc_llc_ops *ops);
```

Again, note that the llc does not handle the physical link. It is thus very easy to mix any physical link with any llc for a given chip driver.

55.1.9 Included Drivers

An HCI based driver for an NXP PN544, connected through I2C bus, and using shdlc is included.

55.1.10 Execution Contexts

The execution contexts are the following: - IRQ handler (IRQH): fast, cannot sleep. sends incoming frames to HCI where they are passed to the current llc. In case of shdlc, the frame is queued in shdlc rx queue.

- SHDLC State Machine worker (SMW)

  Only when llc_shdlc is used: handles shdlc rx & tx queues.

  Dispatches HCI cmd responses.

- HCI Tx Cmd worker (MSGTXWQ)

  Serializes execution of HCI commands.

  Completes execution in case of response timeout.

- HCI Rx worker (MSGRXWQ)

  Dispatches incoming HCI commands or events.

- Syscall context from a userspace call (SYSCALL)

  Any entrypoint in HCI called from NFC Core

55.1.11 Workflow executing an HCI command (using shdlc)

Executing an HCI command can easily be performed synchronously using the following API:

```
int nfc_hci_send_cmd (struct nfc_hci_dev *hdev, u8 gate, u8 cmd,
                      const u8 *param, size_t param_len, struct sk_buff␣
↪**skb)
```

The API must be invoked from a context that can sleep. Most of the time, this will be the syscall context. skb will return the result that was received in the response.

Internally, execution is asynchronous. So all this API does is to enqueue the HCI command, setup a local wait queue on stack, and `wait_event()` for completion. The wait is not interruptible because it is guaranteed that the command will complete after some short timeout anyway.

MSGTXWQ context will then be scheduled and invoke nfc_hci_msg_tx_work(). This function will dequeue the next pending command and send its HCP fragments to the lower layer which happens to be shdlc. It will then start a timer to be able to complete the command with a timeout error if no response arrive.

SMW context gets scheduled and invokes nfc_shdlc_sm_work(). This function handles shdlc framing in and out. It uses the driver xmit to send frames and receives incoming frames in an skb queue filled from the driver IRQ handler. SHDLC I(nformation) frames payload are HCP fragments. They are aggregated to form complete HCI frames, which can be a response, command, or event.

HCI Responses are dispatched immediately from this context to unblock waiting command execution. Response processing involves invoking the completion callback that was provided by nfc_hci_msg_tx_work() when it sent the command. The completion callback will then wake the syscall context.

It is also possible to execute the command asynchronously using this API:

```
static int nfc_hci_execute_cmd_async(struct nfc_hci_dev *hdev, u8 pipe, u8␣
↪cmd,
                                     const u8 *param, size_t param_len,
                                     data_exchange_cb_t cb, void *cb_
↪context)
```

The workflow is the same, except that the API call returns immediately, and the callback will be called with the result from the SMW context.

55.1.12 Workflow receiving an HCI event or command

HCI commands or events are not dispatched from SMW context. Instead, they are queued to HCI rx_queue and will be dispatched from HCI rx worker context (MSGRXWQ). This is done this way to allow a cmd or event handler to also execute other commands (for example, handling the NFC_HCI_EVT_TARGET_DISCOVERED event from PN544 requires to issue an ANY_GET_PARAMETER to the reader A gate to get information on the target that was discovered).

Typically, such an event will be propagated to NFC Core from MSGRXWQ context.

55.1.13 Error management

Errors that occur synchronously with the execution of an NFC Core request are simply returned as the execution result of the request. These are easy.

Errors that occur asynchronously (e.g. in a background protocol handling thread) must be reported such that upper layers don't stay ignorant that something went wrong below and know that expected events will probably never happen. Handling of these errors is done as follows:

- driver (pn544) fails to deliver an incoming frame: it stores the error such that any subsequent call to the driver will result in this error. Then it calls the standard nfc_shdlc_recv_frame() with a NULL argument to report the problem above. shdlc stores a EREMOTEIO sticky status, which will trigger SMW to report above in turn.
- SMW is basically a background thread to handle incoming and outgoing shdlc frames. This thread will also check the shdlc sticky status and report to HCI when it discovers it is not able to run anymore because of an unrecoverable error that happened within shdlc or below. If the problem occurs during shdlc connection, the error is reported through the connect completion.
- HCI: if an internal HCI error happens (frame is lost), or HCI is reported an error from a lower layer, HCI will either complete the currently executing command with that error, or notify NFC Core directly if no command is executing.
- NFC Core: when NFC Core is notified of an error from below and polling is active, it will send a tag discovered event with an empty tag list to the user space to let it know that the poll operation will never be able to detect a tag.

If polling is not active and the error was sticky, lower levels will return it at next invocation.

55.2 Kernel driver for the NXP Semiconductors PN544 Near Field Communication chip

55.2.1 General

The PN544 is an integrated transmission module for contactless communication. The driver goes under drives/nfc/ and is compiled as a module named “pn544”.

Host Interfaces: I2C, SPI and HSU, this driver supports currently only I2C.

55.2.2 Protocols

In the normal (HCI) mode and in the firmware update mode read and write functions behave a bit differently because the message formats or the protocols are different.

In the normal (HCI) mode the protocol used is derived from the ETSI HCI specification. The firmware is updated using a specific protocol, which is different from HCI.

HCI messages consist of an eight bit header and the message body. The header contains the message length. Maximum size for an HCI message is 33. In HCI mode sent messages are tested for a correct checksum. Firmware update messages have the length in the second (MSB) and third (LSB) bytes of the message. The maximum FW message length is 1024 bytes.

For the ETSI HCI specification see http://www.etsi.org/WebSite/Technologies/ProtocolSpecification.aspx

CHAPTER

FIFTYSIX

DMAENGINE DOCUMENTATION

DMAEngine documentation provides documents for various aspects of DMAEngine framework.

56.1 DMAEngine development documentation

This book helps with DMAengine internal APIs and guide for DMAEngine device driver writers.

56.1.1 DMAengine controller documentation

Hardware Introduction

Most of the Slave DMA controllers have the same general principles of operations.

They have a given number of channels to use for the DMA transfers, and a given number of requests lines.

Requests and channels are pretty much orthogonal. Channels can be used to serve several to any requests. To simplify, channels are the entities that will be doing the copy, and requests what endpoints are involved.

The request lines actually correspond to physical lines going from the DMA-eligible devices to the controller itself. Whenever the device will want to start a transfer, it will assert a DMA request (DRQ) by asserting that request line.

A very simple DMA controller would only take into account a single parameter: the transfer size. At each clock cycle, it would transfer a byte of data from one buffer to another, until the transfer size has been reached.

That wouldn’t work well in the real world, since slave devices might require a specific number of bits to be transferred in a single cycle. For example, we may want to transfer as much data as the physical bus allows to maximize performances when doing a simple memory copy operation, but our audio device could have a narrower FIFO that requires data to be written exactly 16 or 24 bits at a time. This is why most if not all of the DMA controllers can adjust this, using a parameter called the transfer width.

Moreover, some DMA controllers, whenever the RAM is used as a source or destination, can group the reads or writes in memory into a buffer, so instead of having a lot of small memory accesses, which is not really efficient, you’ll get several

bigger transfers. This is done using a parameter called the burst size, that defines how many single reads/writes it's allowed to do without the controller splitting the transfer into smaller sub-transfers.

Our theoretical DMA controller would then only be able to do transfers that involve a single contiguous block of data. However, some of the transfers we usually have are not, and want to copy data from non-contiguous buffers to a contiguous buffer, which is called scatter-gather.

DMAEngine, at least for mem2dev transfers, require support for scatter-gather. So we're left with two cases here: either we have a quite simple DMA controller that doesn't support it, and we'll have to implement it in software, or we have a more advanced DMA controller, that implements in hardware scatter-gather.

The latter are usually programmed using a collection of chunks to transfer, and whenever the transfer is started, the controller will go over that collection, doing whatever we programmed there.

This collection is usually either a table or a linked list. You will then push either the address of the table and its number of elements, or the first item of the list to one channel of the DMA controller, and whenever a DRQ will be asserted, it will go through the collection to know where to fetch the data from.

Either way, the format of this collection is completely dependent on your hardware. Each DMA controller will require a different structure, but all of them will require, for every chunk, at least the source and destination addresses, whether it should increment these addresses or not and the three parameters we saw earlier: the burst size, the transfer width and the transfer size.

The one last thing is that usually, slave devices won't issue DRQ by default, and you have to enable this in your slave device driver first whenever you're willing to use DMA.

These were just the general memory-to-memory (also called mem2mem) or memory-to-device (mem2dev) kind of transfers. Most devices often support other kind of transfers or memory operations that dmaengine support and will be detailed later in this document.

DMA Support in Linux

Historically, DMA controller drivers have been implemented using the async TX API, to offload operations such as memory copy, XOR, cryptography, etc., basically any memory to memory operation.

Over time, the need for memory to device transfers arose, and dmaengine was extended. Nowadays, the async TX API is written as a layer on top of dmaengine, and acts as a client. Still, dmaengine accommodates that API in some cases, and made some design choices to ensure that it stayed compatible.

For more information on the Async TX API, please look the relevant documentation file in Documentation/crypto/async-tx-api.txt.

DMAEngine APIs

`struct dma_device` Initialization

Just like any other kernel framework, the whole DMAEngine registration relies on the driver filling a structure and registering against the framework. In our case, that structure is dma_device.

The first thing you need to do in your driver is to allocate this structure. Any of the usual memory allocators will do, but you'll also need to initialize a few fields in there:

- `channels`: should be initialized as a list using the INIT_LIST_HEAD macro for example
- `src_addr_widths`: should contain a bitmask of the supported source transfer width
- `dst_addr_widths`: should contain a bitmask of the supported destination transfer width
- `directions`: should contain a bitmask of the supported slave directions (i.e. excluding mem2mem transfers)
- `residue_granularity`: granularity of the transfer residue reported to dma_set_residue. This can be either:
  - Descriptor: your device doesn't support any kind of residue reporting. The framework will only know that a particular transaction descriptor is done.
  - Segment: your device is able to report which chunks have been transferred
  - Burst: your device is able to report which burst have been transferred
- `dev`: should hold the pointer to the `struct device` associated to your current driver instance.

Supported transaction types

The next thing you need is to set which transaction types your device (and driver) supports.

Our `dma_device structure` has a field called cap_mask that holds the various types of transaction supported, and you need to modify this mask using the dma_cap_set function, with various flags depending on transaction types you support as an argument.

All those capabilities are defined in the `dma_transaction_type enum`, in `include/linux/dmaengine.h`

Currently, the types available are:

- DMA_MEMCPY
  - The device is able to do memory to memory copies

- DMA_XOR
  - The device is able to perform XOR operations on memory areas
  - Used to accelerate XOR intensive tasks, such as RAID5
- DMA_XOR_VAL
  - The device is able to perform parity check using the XOR algorithm against a memory buffer.
- DMA_PQ
  - The device is able to perform RAID6 P+Q computations, P being a simple XOR, and Q being a Reed-Solomon algorithm.
- DMA_PQ_VAL
  - The device is able to perform parity check using RAID6 P+Q algorithm against a memory buffer.
- DMA_INTERRUPT
  - The device is able to trigger a dummy transfer that will generate periodic interrupts
  - Used by the client drivers to register a callback that will be called on a regular basis through the DMA controller interrupt
- DMA_PRIVATE
  - The devices only supports slave transfers, and as such isn't available for async transfers.
- DMA_ASYNC_TX
  - Must not be set by the device, and will be set by the framework if needed
  - TODO: What is it about?
- DMA_SLAVE
  - The device can handle device to memory transfers, including scatter-gather transfers.
  - While in the mem2mem case we were having two distinct types to deal with a single chunk to copy or a collection of them, here, we just have a single transaction type that is supposed to handle both.
  - If you want to transfer a single contiguous memory buffer, simply build a scatter list with only one item.
- DMA_CYCLIC
  - The device can handle cyclic transfers.
  - A cyclic transfer is a transfer where the chunk collection will loop over itself, with the last item pointing to the first.
  - It's usually used for audio transfers, where you want to operate on a single ring buffer that you will fill with your audio data.
- DMA_INTERLEAVE

  - The device supports interleaved transfer.
  - These transfers can transfer data from a non-contiguous buffer to a non-contiguous buffer, opposed to DMA_SLAVE that can transfer data from a non-contiguous data set to a continuous destination buffer.
  - It's usually used for 2d content transfers, in which case you want to transfer a portion of uncompressed data directly to the display to print it

These various types will also affect how the source and destination addresses change over time.

Addresses pointing to RAM are typically incremented (or decremented) after each transfer. In case of a ring buffer, they may loop (DMA_CYCLIC). Addresses pointing to a device's register (e.g. a FIFO) are typically fixed.

Per descriptor metadata support

Some data movement architecture (DMA controller and peripherals) uses metadata associated with a transaction. The DMA controller role is to transfer the payload and the metadata alongside. The metadata itself is not used by the DMA engine itself, but it contains parameters, keys, vectors, etc for peripheral or from the peripheral.

The DMAengine framework provides a generic ways to facilitate the metadata for descriptors. Depending on the architecture the DMA driver can implement either or both of the methods and it is up to the client driver to choose which one to use.

- DESC_METADATA_CLIENT

  The metadata buffer is allocated/provided by the client driver and it is attached (via the dmaengine_desc_attach_metadata() helper to the descriptor.

  From the DMA driver the following is expected for this mode:

  - DMA_MEM_TO_DEV / DEV_MEM_TO_MEM

    The data from the provided metadata buffer should be prepared for the DMA controller to be sent alongside of the payload data. Either by copying to a hardware descriptor, or highly coupled packet.

  - DMA_DEV_TO_MEM

    On transfer completion the DMA driver must copy the metadata to the client provided metadata buffer before notifying the client about the completion. After the transfer completion, DMA drivers must not touch the metadata buffer provided by the client.

- DESC_METADATA_ENGINE

  The metadata buffer is allocated/managed by the DMA driver. The client driver can ask for the pointer, maximum size and the currently used size of the metadata and can directly update or read it. dmaengine_desc_get_metadata_ptr() and dmaengine_desc_set_metadata_len() is provided as helper functions.

  From the DMA driver the following is expected for this mode:

- get_metadata_ptr()

  Should return a pointer for the metadata buffer, the maximum size of the metadata buffer and the currently used / valid (if any) bytes in the buffer.

- set_metadata_len()

  It is called by the clients after it have placed the metadata to the buffer to let the DMA driver know the number of valid bytes provided.

Note: since the client will ask for the metadata pointer in the completion callback (in DMA_DEV_TO_MEM case) the DMA driver must ensure that the descriptor is not freed up prior the callback is called.

Device operations

Our dma_device structure also requires a few function pointers in order to implement the actual logic, now that we described what operations we were able to perform.

The functions that we have to fill in there, and hence have to implement, obviously depend on the transaction types you reported as supported.

- `device_alloc_chan_resources`
- `device_free_chan_resources`
  - These functions will be called whenever a driver will call `dma_request_channel` or `dma_release_channel` for the first/last time on the channel associated to that driver.
  - They are in charge of allocating/freeing all the needed resources in order for that channel to be useful for your driver.
  - These functions can sleep.
- `device_prep_dma_*`
  - These functions are matching the capabilities you registered previously.
  - These functions all take the buffer or the scatterlist relevant for the transfer being prepared, and should create a hardware descriptor or a list of hardware descriptors from it
  - These functions can be called from an interrupt context
  - Any allocation you might do should be using the GFP_NOWAIT flag, in order not to potentially sleep, but without depleting the emergency pool either.
  - Drivers should try to pre-allocate any memory they might need during the transfer setup at probe time to avoid putting to much pressure on the nowait allocator.
  - It should return a unique instance of the `dma_async_tx_descriptor structure`, that further represents this particular transfer.
  - This structure can be initialized using the function `dma_async_tx_descriptor_init`.

  - You’ll also need to set two fields in this structure:
    * flags: TODO: Can it be modified by the driver itself, or should it be always the flags passed in the arguments
    * tx_submit: A pointer to a function you have to implement, that is supposed to push the current transaction descriptor to a pending queue, waiting for issue_pending to be called.
  - In this structure the function pointer callback_result can be initialized in order for the submitter to be notified that a transaction has completed. In the earlier code the function pointer callback has been used. However it does not provide any status to the transaction and will be deprecated. The result structure defined as `dmaengine_result` that is passed in to callback_result has two fields:
    * result: This provides the transfer result defined by `dmaengine_tx_result`. Either success or some error condition.
    * residue: Provides the residue bytes of the transfer for those that support residue.

- `device_issue_pending`
  - Takes the first transaction descriptor in the pending queue, and starts the transfer. Whenever that transfer is done, it should move to the next transaction in the list.
  - This function can be called in an interrupt context
- `device_tx_status`
  - Should report the bytes left to go over on the given channel
  - Should only care about the transaction descriptor passed as argument, not the currently active one on a given channel
  - The tx_state argument might be NULL
  - Should use dma_set_residue to report it
  - In the case of a cyclic transfer, it should only take into account the current period.
  - This function can be called in an interrupt context.
- device_config
  - Reconfigures the channel with the configuration given as argument
  - This command should NOT perform synchronously, or on any currently queued transfers, but only on subsequent ones
  - In this case, the function will receive a `dma_slave_config` structure pointer as an argument, that will detail which configuration to use.
  - Even though that structure contains a direction field, this field is deprecated in favor of the direction argument given to the prep_* functions
  - This call is mandatory for slave operations only. This should NOT be set or expected to be set for memcpy operations. If a driver support both, it should use this call for slave operations only and not for memcpy ones.

- device_pause
  - Pauses a transfer on the channel
  - This command should operate synchronously on the channel, pausing right away the work of the given channel
- device_resume
  - Resumes a transfer on the channel
  - This command should operate synchronously on the channel, resuming right away the work of the given channel
- device_terminate_all
  - Aborts all the pending and ongoing transfers on the channel
  - For aborted transfers the complete callback should not be called
  - Can be called from atomic context or from within a complete callback of a descriptor. Must not sleep. Drivers must be able to handle this correctly.
  - Termination may be asynchronous. The driver does not have to wait until the currently active transfer has completely stopped. See device_synchronize.
- device_synchronize
  - Must synchronize the termination of a channel to the current context.
  - Must make sure that memory for previously submitted descriptors is no longer accessed by the DMA controller.
  - Must make sure that all complete callbacks for previously submitted descriptors have finished running and none are scheduled to run.
  - May sleep.

Misc notes

(stuff that should be documented, but don't really know where to put them)

`dma_run_dependencies`

- Should be called at the end of an async TX transfer, and can be ignored in the slave transfers case.
- Makes sure that dependent operations are run before marking it as complete.

dma_cookie_t

- it's a DMA transaction ID that will increment over time.
- Not really relevant any more since the introduction of `virt-dma` that abstracts it away.

DMA_CTRL_ACK

- If clear, the descriptor cannot be reused by provider until the client acknowledges receipt, i.e. has has a chance to establish any dependency chains
- This can be acked by invoking async_tx_ack()

- If set, does not mean descriptor can be reused

DMA_CTRL_REUSE

- If set, the descriptor can be reused after being completed. It should not be freed by provider if this flag is set.
- The descriptor should be prepared for reuse by invoking `dmaengine_desc_set_reuse()` which will set DMA_CTRL_REUSE.
- `dmaengine_desc_set_reuse()` will succeed only when channel support reusable descriptor as exhibited by capabilities
- As a consequence, if a device driver wants to skip the `dma_map_sg()` and `dma_unmap_sg()` in between 2 transfers, because the DMA'd data wasn't used, it can resubmit the transfer right after its completion.
- Descriptor can be freed in few ways
  - Clearing DMA_CTRL_REUSE by invoking `dmaengine_desc_clear_reuse()` and submitting for last txn
  - Explicitly invoking `dmaengine_desc_free()`, this can succeed only when DMA_CTRL_REUSE is already set
  - Terminating the channel
- DMA_PREP_CMD
  - If set, the client driver tells DMA controller that passed data in DMA API is command data.
  - Interpretation of command data is DMA controller specific. It can be used for issuing commands to other peripherals/register reads/register writes for which the descriptor should be in different format from normal data descriptors.

General Design Notes

Most of the DMAEngine drivers you'll see are based on a similar design that handles the end of transfer interrupts in the handler, but defer most work to a tasklet, including the start of a new transfer whenever the previous transfer ended.

This is a rather inefficient design though, because the inter-transfer latency will be not only the interrupt latency, but also the scheduling latency of the tasklet, which will leave the channel idle in between, which will slow down the global transfer rate.

You should avoid this kind of practice, and instead of electing a new transfer in your tasklet, move that part to the interrupt handler in order to have a shorter idle window (that we can't really avoid anyway).

Glossary

- Burst: A number of consecutive read or write operations that can be queued to buffers before being flushed to memory.
- Chunk: A contiguous collection of bursts
- Transfer: A collection of chunks (be it contiguous or not)

56.2 DMAEngine client documentation

This book is a guide to device driver writers on how to use the Slave-DMA API of the DMAEngine. This is applicable only for slave DMA usage only.

56.2.1 DMA Engine API Guide

Vinod Koul <vinod dot koul at intel.com>

Note: For DMA Engine usage in async_tx please see: `Documentation/crypto/async-tx-api.txt`

Below is a guide to device driver writers on how to use the Slave-DMA API of the DMA Engine. This is applicable only for slave DMA usage only.

DMA usage

The slave DMA usage consists of following steps:

- Allocate a DMA slave channel
- Set slave and controller specific parameters
- Get a descriptor for transaction
- Submit the transaction
- Issue pending requests and wait for callback notification

The details of these operations are:

1. Allocate a DMA slave channel

   Channel allocation is slightly different in the slave DMA context, client drivers typically need a channel from a particular DMA controller only and even in some cases a specific channel is desired. To request a channel dma_request_chan() API is used.

   Interface:

```
struct dma_chan *dma_request_chan(struct device *dev, const char␣
↪*name);
```

Which will find and return the `name` DMA channel associated with the 'dev' device. The association is done via DT, ACPI or board file based dma_slave_map matching table.

A channel allocated via this interface is exclusive to the caller, until dma_release_channel() is called.

2. Set slave and controller specific parameters

   Next step is always to pass some specific information to the DMA driver. Most of the generic information which a slave DMA can use is in struct dma_slave_config. This allows the clients to specify DMA direction, DMA addresses, bus widths, DMA burst lengths etc for the peripheral.

   If some DMA controllers have more parameters to be sent then they should try to embed struct dma_slave_config in their controller specific structure. That gives flexibility to client to pass more parameters, if required.

   Interface:

```
int dmaengine_slave_config(struct dma_chan *chan,
                   struct dma_slave_config *config)
```

   Please see the dma_slave_config structure definition in dmaengine.h for a detailed explanation of the struct members. Please note that the 'direction' member will be going away as it duplicates the direction given in the prepare call.

3. Get a descriptor for transaction

   For slave usage the various modes of slave transfers supported by the DMA-engine are:

   - slave_sg: DMA a list of scatter gather buffers from/to a peripheral
   - dma_cyclic: Perform a cyclic DMA operation from/to a peripheral till the operation is explicitly stopped.
   - interleaved_dma: This is common to Slave as well as M2M clients. For slave address of devices' fifo could be already known to the driver. Various types of operations could be expressed by setting appropriate values to the 'dma_interleaved_template' members.

   A non-NULL return of this transfer API represents a "descriptor" for the given transaction.

   Interface:

```
struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(
           struct dma_chan *chan, struct scatterlist *sgl,
           unsigned int sg_len, enum dma_data_direction direction,
           unsigned long flags);

struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(
           struct dma_chan *chan, dma_addr_t buf_addr, size_t buf_
↪len,
           size_t period_len, enum dma_data_direction direction);
```

(continues on next page)

(continued from previous page)

```
struct dma_async_tx_descriptor *dmaengine_prep_interleaved_dma(
          struct dma_chan *chan, struct dma_interleaved_template␣
↪*xt,
          unsigned long flags);
```

The peripheral driver is expected to have mapped the scatterlist for the DMA operation prior to calling dmaengine_prep_slave_sg(), and must keep the scatterlist mapped until the DMA operation has completed. The scatterlist must be mapped using the DMA struct device. If a mapping needs to be synchronized later, dma_sync_*_for_*() must be called using the DMA struct device, too. So, normal setup should look like this:

```
nr_sg = dma_map_sg(chan->device->dev, sgl, sg_len);
   if (nr_sg == 0)
          /* error */

   desc = dmaengine_prep_slave_sg(chan, sgl, nr_sg, direction,␣
↪flags);
```

Once a descriptor has been obtained, the callback information can be added and the descriptor must then be submitted. Some DMA engine drivers may hold a spinlock between a successful preparation and submission so it is important that these two operations are closely paired.

Note: Although the async_tx API specifies that completion callback routines cannot submit any new operations, this is not the case for slave/cyclic DMA.

For slave DMA, the subsequent transaction may not be available for submission prior to callback function being invoked, so slave DMA callbacks are permitted to prepare and submit a new transaction.

For cyclic DMA, a callback function may wish to terminate the DMA via dmaengine_terminate_async().

Therefore, it is important that DMA engine drivers drop any locks before calling the callback function which may cause a deadlock.

Note that callbacks will always be invoked from the DMA engines tasklet, never from interrupt context.

Optional: per descriptor metadata

DMAengine provides two ways for metadata support.

DESC_METADATA_CLIENT

> The metadata buffer is allocated/provided by the client driver and it is attached to the descriptor.

```
int dmaengine_desc_attach_metadata(struct dma_async_tx_descriptor␣
↪*desc,
                                   void *data, size_t len);
```

DESC_METADATA_ENGINE

The metadata buffer is allocated/managed by the DMA driver. The client driver can ask for the pointer, maximum size and the currently used size of the metadata and can directly update or read it.

Becasue the DMA driver manages the memory area containing the metadata, clients must make sure that they do not try to access or get the pointer after their transfer completion callback has run for the descriptor. If no completion callback has been defined for the transfer, then the metadata must not be accessed after issue_pending. In other words: if the aim is to read back metadata after the transfer is completed, then the client must use completion callback.

```
void *dmaengine_desc_get_metadata_ptr(struct dma_async_tx_
↪descriptor *desc,
          size_t *payload_len, size_t *max_len);

int dmaengine_desc_set_metadata_len(struct dma_async_tx_
↪descriptor *desc,
          size_t payload_len);
```

Client drivers can query if a given mode is supported with:

```
bool dmaengine_is_metadata_mode_supported(struct dma_chan *chan,
          enum dma_desc_metadata_mode mode);
```

Depending on the used mode client drivers must follow different flow.

DESC_METADATA_CLIENT

- DMA_MEM_TO_DEV / DEV_MEM_TO_MEM:
  1. prepare the descriptor (dmaengine_prep_*) construct the metadata in the client's buffer
  2. use dmaengine_desc_attach_metadata() to attach the buffer to the descriptor
  3. submit the transfer
- DMA_DEV_TO_MEM:
  1. prepare the descriptor (dmaengine_prep_*)
  2. use dmaengine_desc_attach_metadata() to attach the buffer to the descriptor
  3. submit the transfer
  4. when the transfer is completed, the metadata should be available in the attached buffer

DESC_METADATA_ENGINE

- DMA_MEM_TO_DEV / DEV_MEM_TO_MEM:
  1. prepare the descriptor (dmaengine_prep_*)

2. use dmaengine_desc_get_metadata_ptr() to get the pointer to the engine's metadata area
3. update the metadata at the pointer
4. use dmaengine_desc_set_metadata_len() to tell the DMA engine the amount of data the client has placed into the metadata buffer
5. submit the transfer

- DMA_DEV_TO_MEM:
  1. prepare the descriptor (dmaengine_prep_*)
  2. submit the transfer
  3. on transfer completion, use dmaengine_desc_get_metadata_ptr() to get the pointer to the engine's metadata area
  4. read out the metadata from the pointer

Note: When DESC_METADATA_ENGINE mode is used the metadata area for the descriptor is no longer valid after the transfer has been completed (valid up to the point when the completion callback returns if used).

Mixed use of DESC_METADATA_CLIENT / DESC_METADATA_ENGINE is not allowed, client drivers must use either of the modes per descriptor.

---

4. Submit the transaction

   Once the descriptor has been prepared and the callback information added, it must be placed on the DMA engine drivers pending queue.

   Interface:

```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc)
```

   This returns a cookie can be used to check the progress of DMA engine activity via other DMA engine calls not covered in this document.

   dmaengine_submit() will not start the DMA operation, it merely adds it to the pending queue. For this, see step 5, dma_async_issue_pending.

Note: After calling `dmaengine_submit()` the submitted transfer descriptor (`struct dma_async_tx_descriptor`) belongs to the DMA engine. Consequently, the client must consider invalid the pointer to that descriptor.

---

5. Issue pending DMA requests and wait for callback notification

   The transactions in the pending queue can be activated by calling the issue_pending API. If channel is idle then the first transaction in queue is started and subsequent ones queued up.

   On completion of each DMA operation, the next in queue is started and a tasklet triggered. The tasklet will then call the client driver completion callback routine for notification, if set.

Interface:

```
void dma_async_issue_pending(struct dma_chan *chan);
```

Further APIs

1. Terminate APIs

```
int dmaengine_terminate_sync(struct dma_chan *chan)
int dmaengine_terminate_async(struct dma_chan *chan)
int dmaengine_terminate_all(struct dma_chan *chan) /* DEPRECATED */
```

This causes all activity for the DMA channel to be stopped, and may discard data in the DMA FIFO which hasn't been fully transferred. No callback functions will be called for any incomplete transfers.

Two variants of this function are available.

dmaengine_terminate_async() might not wait until the DMA has been fully stopped or until any running complete callbacks have finished. But it is possible to call dmaengine_terminate_async() from atomic context or from within a complete callback. dmaengine_synchronize() must be called before it is safe to free the memory accessed by the DMA transfer or free resources accessed from within the complete callback.

dmaengine_terminate_sync() will wait for the transfer and any running complete callbacks to finish before it returns. But the function must not be called from atomic context or from within a complete callback.

dmaengine_terminate_all() is deprecated and should not be used in new code.

2. Pause API

```
int dmaengine_pause(struct dma_chan *chan)
```

This pauses activity on the DMA channel without data loss.

3. Resume API

```
int dmaengine_resume(struct dma_chan *chan)
```

Resume a previously paused DMA channel. It is invalid to resume a channel which is not currently paused.

4. Check Txn complete

```
enum dma_status dma_async_is_tx_complete(struct dma_chan *chan,
          dma_cookie_t cookie, dma_cookie_t *last, dma_cookie_t *used)
```

This can be used to check the status of the channel. Please see the documentation in include/linux/dmaengine.h for a more complete description of this API.

This can be used in conjunction with dma_async_is_complete() and the cookie returned from dmaengine_submit() to check for completion of a specific DMA transaction.

Note: Not all DMA engine drivers can return reliable information for a running DMA channel. It is recommended that DMA engine users pause or stop (via dmaengine_terminate_all()) the channel before using this API.

5. Synchronize termination API

```
void dmaengine_synchronize(struct dma_chan *chan)
```

Synchronize the termination of the DMA channel to the current context.

This function should be used after dmaengine_terminate_async() to synchronize the termination of the DMA channel to the current context. The function will wait for the transfer and any running complete callbacks to finish before it returns.

If dmaengine_terminate_async() is used to stop the DMA channel this function must be called before it is safe to free memory accessed by previously submitted descriptors or to free any resources accessed within the complete callback of previously submitted descriptors.

The behavior of this function is undefined if dma_async_issue_pending() has been called between dmaengine_terminate_async() and this function.

56.3 DMA Test documentation

This book introduces how to test DMA drivers using dmatest module.

56.3.1 DMA Test Guide

Andy Shevchenko <andriy.shevchenko@linux.intel.com>

This small document introduces how to test DMA drivers using dmatest module.

Note: The test suite works only on the channels that have at least one capability of the following: DMA_MEMCPY (memory-to-memory), DMA_MEMSET (const-to-memory or memory-to-memory, when emulated), DMA_XOR, DMA_PQ.

Note: In case of any related questions use the official mailing list dmaengine@vger.kernel.org.

Part 1 - How to build the test module

The menuconfig contains an option that could be found by following path:

Device Drivers -> DMA Engine support -> DMA Test client

In the configuration file the option called CONFIG_DMATEST. The dmatest could be built as module or inside kernel. Let's consider those cases.

Part 2 - When dmatest is built as a module

Example of usage:

```
% modprobe dmatest timeout=2000 iterations=1 channel=dma0chan0 run=1
```

…or:

```
% modprobe dmatest
% echo 2000 > /sys/module/dmatest/parameters/timeout
% echo 1 > /sys/module/dmatest/parameters/iterations
% echo dma0chan0 > /sys/module/dmatest/parameters/channel
% echo 1 > /sys/module/dmatest/parameters/run
```

…or on the kernel command line:

```
dmatest.timeout=2000 dmatest.iterations=1 dmatest.channel=dma0chan0␣
↪dmatest.run=1
```

Example of multi-channel test usage (new in the 5.0 kernel):

```
% modprobe dmatest
% echo 2000 > /sys/module/dmatest/parameters/timeout
% echo 1 > /sys/module/dmatest/parameters/iterations
% echo dma0chan0 > /sys/module/dmatest/parameters/channel
% echo dma0chan1 > /sys/module/dmatest/parameters/channel
% echo dma0chan2 > /sys/module/dmatest/parameters/channel
% echo 1 > /sys/module/dmatest/parameters/run
```

Note: For all tests, starting in the 5.0 kernel, either single- or multi-channel, the channel parameter(s) must be set after all other parameters. It is at that time that the existing parameter values are acquired for use by the thread(s). All other parameters are shared. Therefore, if changes are made to any of the other parameters, and an additional channel specified, the (shared) parameters used for all threads will use the new values. After the channels are specified, each thread is set as pending. All threads begin execution when the run parameter is set to 1.

Hint: A list of available channels can be found by running the following command:

```
% ls -1 /sys/class/dma/
```

Once started a message like ” dmatest: Added 1 threads using dma0chan0” is emitted. A thread for that specific channel is created and is now pending, the pending thread is started once run is to 1.

Note that running a new test will not stop any in progress test.

The following command returns the state of the test.

```
% cat /sys/module/dmatest/parameters/run
```

To wait for test completion userpace can poll ‘run’ until it is false, or use the wait parameter. Specifying ‘wait=1’ when loading the module causes module initialization to pause until a test run has completed, while reading /sys/module/dmatest/parameters/wait waits for any running test to complete before returning. For example, the following scripts wait for 42 tests to complete before exiting. Note that if ‘iterations’ is set to ‘infinite’ then waiting is disabled.

Example:

```
% modprobe dmatest run=1 iterations=42 wait=1
% modprobe -r dmatest
```

⋯or:

```
% modprobe dmatest run=1 iterations=42
% cat /sys/module/dmatest/parameters/wait
% modprobe -r dmatest
```

Part 3 - When built-in in the kernel

The module parameters that is supplied to the kernel command line will be used for the first performed test. After user gets a control, the test could be re-run with the same or different parameters. For the details see the above section Part 2 - When dmatest is built as a module.

In both cases the module parameters are used as the actual values for the test case. You always could check them at run-time by running

```
% grep -H . /sys/module/dmatest/parameters/*
```

Part 4 - Gathering the test results

Test results are printed to the kernel log buffer with the format:

```
"dmatest: result <channel>: <test id>: '<error msg>' with src_off=<val>␣
↪dst_off=<val> len=<val> (<err code>)"
```

Example of output:

```
% dmesg | tail -n 1
dmatest: result dma0chan0-copy0: #1: No errors with src_off=0x7bf dst_
↪off=0x8ad len=0x3fea (0)
```

The message format is unified across the different types of errors. A number in the parentheses represents additional information, e.g. error code, error counter, or status. A test thread also emits a summary line at completion listing the number of tests executed, number that failed, and a result code.

Example:

```
% dmesg | tail -n 1
dmatest: dma0chan0-copy0: summary 1 test, 0 failures 1000 iops 100000 KB/s␣
↪(0)
```

The details of a data miscompare error are also emitted, but do not follow the above format.

Part 5 - Handling channel allocation

Allocating Channels

Channels are required to be configured prior to starting the test run. Attempting to run the test without configuring the channels will fail.

Example:

```
% echo 1 > /sys/module/dmatest/parameters/run
dmatest: Could not start test, no channels configured
```

Channels are registered using the “channel”parameter. Channels can be requested by their name, once requested, the channel is registered and a pending thread is added to the test list.

Example:

```
% echo dma0chan2 > /sys/module/dmatest/parameters/channel
dmatest: Added 1 threads using dma0chan2
```

More channels can be added by repeating the example above. Reading back the channel parameter will return the name of last channel that was added successfully.

Example:

```
% echo dma0chan1 > /sys/module/dmatest/parameters/channel
dmatest: Added 1 threads using dma0chan1
% echo dma0chan2 > /sys/module/dmatest/parameters/channel
dmatest: Added 1 threads using dma0chan2
% cat /sys/module/dmatest/parameters/channel
dma0chan2
```

Another method of requesting channels is to request a channel with an empty string, Doing so will request all channels available to be tested:

Example:

```
% echo "" > /sys/module/dmatest/parameters/channel
dmatest: Added 1 threads using dma0chan0
```

(continues on next page)

(continued from previous page)

```
dmatest: Added 1 threads using dma0chan3
dmatest: Added 1 threads using dma0chan4
dmatest: Added 1 threads using dma0chan5
dmatest: Added 1 threads using dma0chan6
dmatest: Added 1 threads using dma0chan7
dmatest: Added 1 threads using dma0chan8
```

At any point during the test configuration, reading the "test_list" parameter will print the list of currently pending tests.

Example:

```
% cat /sys/module/dmatest/parameters/test_list
dmatest: 1 threads using dma0chan0
dmatest: 1 threads using dma0chan3
dmatest: 1 threads using dma0chan4
dmatest: 1 threads using dma0chan5
dmatest: 1 threads using dma0chan6
dmatest: 1 threads using dma0chan7
dmatest: 1 threads using dma0chan8
```

Note: Channels will have to be configured for each test run as channel configurations do not carry across to the next test run.

Releasing Channels

Channels can be freed by setting run to 0.

Example:

```
% echo dma0chan1 > /sys/module/dmatest/parameters/channel
dmatest: Added 1 threads using dma0chan1
% cat /sys/class/dma/dma0chan1/in_use
1
% echo 0 > /sys/module/dmatest/parameters/run
% cat /sys/class/dma/dma0chan1/in_use
0
```

Channels allocated by previous test runs are automatically freed when a new channel is requested after completing a successful test run.

56.4 PXA DMA documentation

This book adds some notes about PXA DMA

56.4.1 PXA/MMP - DMA Slave controller

Constraints

a) Transfers hot queuing A driver submitting a transfer and issuing it should be granted the transfer is queued even on a running DMA channel. This implies that the queuing doesn't wait for the previous transfer end, and that the descriptor chaining is not only done in the irq/tasklet code triggered by the end of the transfer. A transfer which is submitted and issued on a phy doesn't wait for a phy to stop and restart, but is submitted on a "running channel". The other drivers, especially mmp_pdma waited for the phy to stop before relaunching a new transfer.

b) All transfers having asked for confirmation should be signaled Any issued transfer with DMA_PREP_INTERRUPT should trigger a callback call. This implies that even if an irq/tasklet is triggered by end of tx1, but at the time of irq/dma tx2 is already finished, tx1->complete() and tx2->complete() should be called.

c) Channel running state A driver should be able to query if a channel is running or not. For the multimedia case, such as video capture, if a transfer is submitted and then a check of the DMA channel reports a "stopped channel", the transfer should not be issued until the next "start of frame interrupt", hence the need to know if a channel is in running or stopped state.

d) Bandwidth guarantee The PXA architecture has 4 levels of DMAs priorities : high, normal, low. The high priorities get twice as much bandwidth as the normal, which get twice as much as the low priorities. A driver should be able to request a priority, especially the real-time ones such as pxa_camera with (big) throughputs.

Design

a) Virtual channels Same concept as in sa11x0 driver, ie. a driver was assigned a "virtual channel" linked to the requestor line, and the physical DMA channel is assigned on the fly when the transfer is issued.

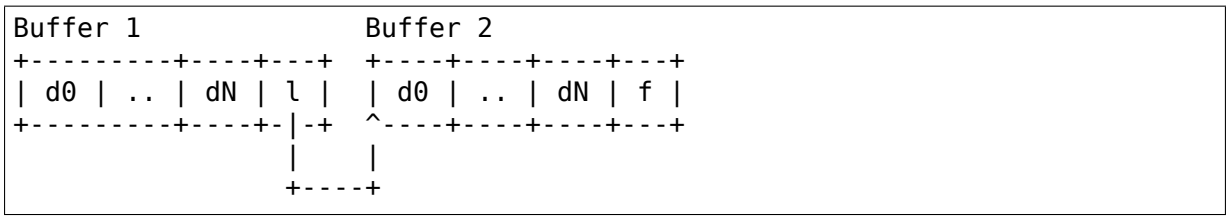
b) Transfer anatomy for a scatter-gather transfer

```
+------------+-----+--------------+----------------+-----------------+
| desc-sg[0] | ... | desc-sg[last] | status updater | finisher/linker |
+------------+-----+--------------+----------------+-----------------+
```

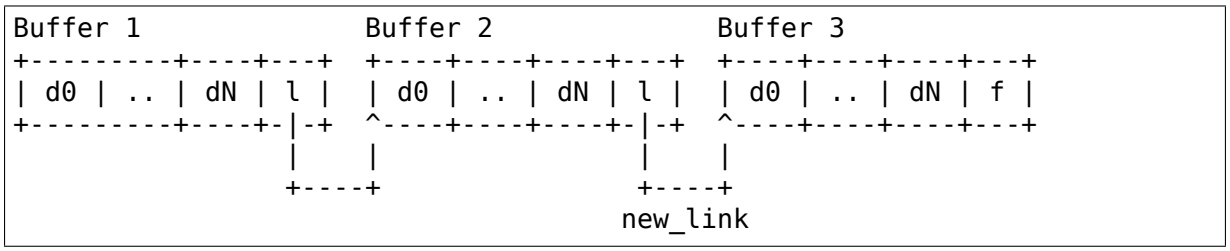
This structure is pointed by dma->sg_cpu. The descriptors are used as follows :

- desc-sg[i]: i-th descriptor, transferring the i-th sg element to the video buffer scatter gather
- status updater Transfers a single u32 to a well known dma coherent memory to leave a trace that this transfer is done. The "well known" is unique per physical channel, meaning that a read of this value will tell which is the last finished transfer at that point in time.
- finisher: has ddadr=DADDR_STOP, dcmd=ENDIRQEN
- linker: has ddadr= desc-sg[0] of next transfer, dcmd=0

c) Transfers hot-chaining Suppose the running chain is:



After a call to dmaengine_submit(b3), the chain will look like:



If while new_link was created the DMA channel stopped, it is _not_ restarted. Hot-chaining doesn't break the assumption that dma_async_issue_pending() is to be used to ensure the transfer is actually started.

One exception to this rule :

- if Buffer1 and Buffer2 had all their addresses 8 bytes aligned
- and if Buffer3 has at least one address not 4 bytes aligned
- then hot-chaining cannot happen, as the channel must be stopped, the "align bit" must be set, and the channel restarted As a consequence, such a transfer tx_submit() will be queued on the submitted queue, and this specific case if the DMA is already running in aligned mode.

d) Transfers completion updater Each time a transfer is completed on a channel, an interrupt might be generated or not, up to the client's request. But in each case, the last descriptor of a transfer, the "status updater", will write the latest transfer being completed into the physical channel's completion mark.

This will speed up residue calculation, for large transfers such as video buffers which hold around 6k descriptors or more. This also allows without any lock to find out what is the latest completed transfer in a running DMA chain.

e) Transfers completion, irq and tasklet When a transfer flagged as "DMA_PREP_INTERRUPT" is finished, the dma irq is raised. Upon this interrupt, a tasklet is scheduled for the physical channel.

The tasklet is responsible for :

- reading the physical channel last updater mark
- calling all the transfer callbacks of finished transfers, based on that mark, and each transfer flags.

If a transfer is completed while this handling is done, a dma irq will be raised, and the tasklet will be scheduled once again, having a new updater mark.

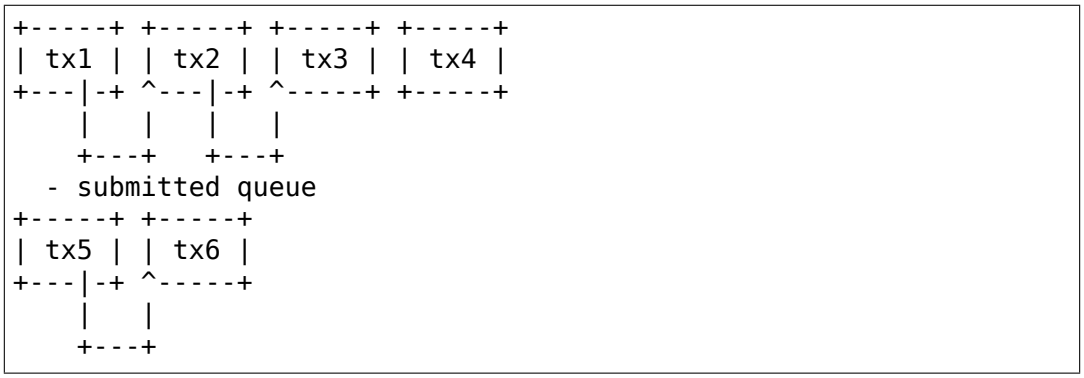
f) Residue Residue granularity will be descriptor based. The issued but not completed transfers will be scanned for all of their descriptors against the currently running descriptor.

g) Most complicated case of driver’s tx queues The most tricky situation is when :

- there are not “acked” transfers (tx0)
- a driver submitted an aligned tx1, not chained
- a driver submitted an aligned tx2 => tx2 is cold chained to tx1
- a driver issued tx1+tx2 => channel is running in aligned mode
- a driver submitted an aligned tx3 => tx3 is hot-chained
- a driver submitted an unaligned tx4 => tx4 is put in submitted queue, not chained
- a driver issued tx4 => tx4 is put in issued queue, not chained
- a driver submitted an aligned tx5 => tx5 is put in submitted queue, not chained
- a driver submitted an aligned tx6 => tx6 is put in submitted queue, cold chained to tx5

This translates into (after tx4 is issued) :

- issued queue



- completed queue : empty
- allocated queue : tx0

It should be noted that after tx3 is completed, the channel is stopped, and restarted in “unaligned mode” to handle tx4.

Author: Robert Jarzmik <robert.jarzmik@free.fr>

CHAPTER

FIFTYSEVEN

LINUX KERNEL SLIMBUS SUPPORT

57.1 Overview

57.1.1 What is SLIMbus?

SLIMbus (Serial Low Power Interchip Media Bus) is a specification developed by MIPI (Mobile Industry Processor Interface) alliance. The bus uses master/slave configuration, and is a 2-wire multi-drop implementation (clock, and data).

Currently, SLIMbus is used to interface between application processors of SoCs (System-on-Chip) and peripheral components (typically codec). SLIMbus uses Time-Division-Multiplexing to accommodate multiple data channels, and a control channel.

The control channel is used for various control functions such as bus management, configuration and status updates. These messages can be unicast (e.g. reading/writing device specific values), or multicast (e.g. data channel reconfiguration sequence is a broadcast message announced to all devices)

A data channel is used for data-transfer between 2 SLIMbus devices. Data channel uses dedicated ports on the device.

57.1.2 Hardware description:

SLIMbus specification has different types of device classifications based on their capabilities. A manager device is responsible for enumeration, configuration, and dynamic channel allocation. Every bus has 1 active manager.

A generic device is a device providing application functionality (e.g. codec).

Framer device is responsible for clocking the bus, and transmitting frame-sync and framing information on the bus.

Each SLIMbus component has an interface device for monitoring physical layer.

Typically each SoC contains SLIMbus component having 1 manager, 1 framer device, 1 generic device (for data channel support), and 1 interface device. External peripheral SLIMbus component usually has 1 generic device (for functionality/data channel support), and an associated interface device. The generic device’s registers are mapped as ‘value elements’ so that they can be written/read using SLIMbus control channel exchanging control/status type of information. In case

there are multiple framer devices on the same bus, manager device is responsible to select the active-framer for clocking the bus.

Per specification, SLIMbus uses "clock gears" to do power management based on current frequency and bandwidth requirements. There are 10 clock gears and each gear changes the SLIMbus frequency to be twice its previous gear.

Each device has a 6-byte enumeration-address and the manager assigns every device with a 1-byte logical address after the devices report presence on the bus.

57.1.3 Software description:

There are 2 types of SLIMbus drivers:

slim_controller represents a 'controller' for SLIMbus. This driver should implement duties needed by the SoC (manager device, associated interface device for monitoring the layers and reporting errors, default framer device).

slim_device represents the 'generic device/component' for SLIMbus, and a slim_driver should implement driver for that slim_device.

57.1.4 Device notifications to the driver:

Since SLIMbus devices have mechanisms for reporting their presence, the framework allows drivers to bind when corresponding devices report their presence on the bus. However, it is possible that the driver needs to be probed first so that it can enable corresponding SLIMbus device (e.g. power it up and/or take it out of reset). To support that behavior, the framework allows drivers to probe first as well (e.g. using standard DeviceTree compatibility field). This creates the necessity for the driver to know when the device is functional (i.e. reported present). device_up callback is used for that reason when the device reports present and is assigned a logical address by the controller.

Similarly, SLIMbus devices 'report absent' when they go down. A 'device_down' callback notifies the driver when the device reports absent and its logical address assignment is invalidated by the controller.

Another notification "boot_device" is used to notify the slim_driver when controller resets the bus. This notification allows the driver to take necessary steps to boot the device so that it' s functional after the bus has been reset.

57.1.5 Driver and Controller APIs:

struct **slim_eaddr**
    Enumeration address for a SLIMbus device

Definition

```
struct slim_eaddr {
  u8 instance;
  u8 dev_index;
  u16 prod_code;
```

(continues on next page)

(continued from previous page)

```
  u16 manf_id;
};
```

Members

instance Instance value

dev_index Device index

prod_code Product code

manf_id Manufacturer Id for the device

enum **slim_device_status**
slim device status

Constants

SLIM_DEVICE_STATUS_DOWN Slim device is absent or not reported yet.

SLIM_DEVICE_STATUS_UP Slim device is announced on the bus.

SLIM_DEVICE_STATUS_RESERVED Reserved for future use.

struct **slim_device**
Slim device handle.

Definition

```
struct slim_device {
  struct device           dev;
  struct slim_eaddr       e_addr;
  struct slim_controller  *ctrl;
  enum slim_device_status status;
  u8 laddr;
  bool is_laddr_valid;
  struct list_head        stream_list;
  spinlock_t stream_list_lock;
};
```

Members

dev Driver model representation of the device.

e_addr Enumeration address of this device.

ctrl slim controller instance.

status slim device status

laddr 1-byte Logical address of this device.

is_laddr_valid indicates if the laddr is valid or not

stream_list List of streams on this device

stream_list_lock lock to protect the stream list

Description

This is the client/device handle returned when a SLIMbus device is registered with a controller. Pointer to this structure is used by client-driver as a handle.

struct **slim_driver**
SLIMbus 'generic device' (slave) device driver (similar to 'spi_device' on SPI)

Definition

```
struct slim_driver {
  int (*probe)(struct slim_device *sl);
  void (*remove)(struct slim_device *sl);
  void (*shutdown)(struct slim_device *sl);
  int (*device_status)(struct slim_device *sl, enum slim_device_status s);
  struct device_driver            driver;
  const struct slim_device_id     *id_table;
};
```

Members

probe Binds this driver to a SLIMbus device.

remove Unbinds this driver from the SLIMbus device.

shutdown Standard shutdown callback used during powerdown/halt.

device_status This callback is called when - The device reports present and gets a laddr assigned - The device reports absent, or the bus goes down.

driver SLIMbus device drivers should initialize name and owner field of this structure

id_table List of SLIMbus devices supported by this driver

struct **slim_val_inf**
Slimbus value or information element

Definition

```
struct slim_val_inf {
  u16 start_offset;
  u8 num_bytes;
  u8 *rbuf;
  const u8                *wbuf;
  struct completion      *comp;
};
```

Members

start_offset Specifies starting offset in information/value element map

num_bytes upto 16. This ensures that the message will fit the slicesize per SLIMbus spec

rbuf buffer to read the values

wbuf buffer to write

comp completion for asynchronous operations, valid only if TID is required for transaction, like REQUEST operations. Rest of the transactions are synchronous anyway.

struct **slim_stream_config**
SLIMbus stream configuration Configuring a stream is done at hw_params or

prepare call from audio drivers where they have all the required information regarding rate, number of channels and so on. There is a 1:1 mapping of channel and ports.

Definition

```
struct slim_stream_config {
  unsigned int rate;
  unsigned int bps;
  unsigned int ch_count;
  unsigned int *chs;
  unsigned long port_mask;
  int direction;
};
```

Members

rate data rate

bps bits per data sample

ch_count number of channels

chs pointer to list of channel numbers

port_mask port mask of ports to use for this stream

direction direction of the stream, SNDRV_PCM_STREAM_PLAYBACK or SNDRV_PCM_STREAM_CAPTURE.

module_slim_driver(__slim_driver)
Helper macro for registering a SLIMbus driver

Parameters

__slim_driver slimbus_driver struct

Description

Helper macro for SLIMbus drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init()` and `module_exit()`

struct **slim_framer**
Represents SLIMbus framer. Every controller may have multiple framers. There is 1 active framer device responsible for clocking the bus. Manager is responsible for framer hand-over.

Definition

```
struct slim_framer {
  struct device           dev;
  struct slim_eaddr       e_addr;
  int rootfreq;
  int superfreq;
};
```

Members

dev Driver model representation of the device.

e_addr Enumeration address of the framer.

rootfreq Root Frequency at which the framer can run. This is maximum frequency ('clock gear 10') at which the bus can operate.

superfreq Superframes per root frequency. Every frame is 6144 bits.

struct **slim_msg_txn**
Message to be sent by the controller. This structure has packet header, payload and buffer to be filled (if any)

Definition

```
struct slim_msg_txn {
  u8 rl;
  u8 mt;
  u8 mc;
  u8 dt;
  u16 ec;
  u8 tid;
  u8 la;
  struct slim_val_inf     *msg;
  struct completion      *comp;
};
```

Members

rl Header field. remaining length.

mt Header field. Message type.

mc Header field. LSB is message code for type mt.

dt Header field. Destination type.

ec Element code. Used for elemental access APIs.

tid Transaction ID. Used for messages expecting response. (relevant for message-codes involving read operation)

la Logical address of the device this message is going to. (Not used when destination type is broadcast.)

msg Elemental access message to be read/written

comp completion if read/write is synchronous, used internally for tid based transactions.

enum **slim_clk_state**

Constants

SLIM_CLK_ACTIVE SLIMbus clock is active

SLIM_CLK_ENTERING_PAUSE SLIMbus clock pause sequence is being sent on the bus. If this succeeds, state changes to SLIM_CLK_PAUSED. If the transition fails, state changes back to SLIM_CLK_ACTIVE

SLIM_CLK_PAUSED SLIMbus controller clock has paused.

Description

maintaining current clock state.

struct **slim_sched**

Definition

```
struct slim_sched {
  enum slim_clk_state     clk_state;
  struct completion       pause_comp;
  struct mutex            m_reconf;
};
```

Members

clk_state Controller's clock state from enum slim_clk_state

pause_comp Signals completion of clock pause sequence. This is useful when client tries to call SLIMbus transaction when controller is entering clock pause.

m_reconf This mutex is held until current reconfiguration (data channel scheduling, message bandwidth reservation) is done. Message APIs can use the bus concurrently when this mutex is held since elemental access messages can be sent on the bus when reconfiguration is in progress.

enum **slim_port_direction**

Constants

SLIM_PORT_SINK SLIMbus port is a sink

SLIM_PORT_SOURCE SLIMbus port is a source

enum **slim_port_state**

Constants

SLIM_PORT_DISCONNECTED SLIMbus port is disconnected entered from Unconfigure/configured state after DISCONNECT_PORT or REMOVE_CHANNEL core command

SLIM_PORT_UNCONFIGURED SLIMbus port is in unconfigured state. entered from disconnect state after CONNECT_SOURCE/SINK core command

SLIM_PORT_CONFIGURED SLIMbus port is in configured state. entered from unconfigured state after DEFINE_CHANNEL, DEFINE_CONTENT and ACTIVATE_CHANNEL core commands. Ready for data transmission.

Description

according to SLIMbus Spec 2.0

enum **slim_channel_state**

Constants

SLIM_CH_STATE_DISCONNECTED SLIMbus channel is disconnected

SLIM_CH_STATE_ALLOCATED SLIMbus channel is allocated

SLIM_CH_STATE_ASSOCIATED SLIMbus channel is associated with port

SLIM_CH_STATE_DEFINED SLIMbus channel parameters are defined

SLIM_CH_STATE_CONTENT_DEFINED SLIMbus channel content is defined

SLIM_CH_STATE_ACTIVE SLIMbus channel is active and ready for data

SLIM_CH_STATE_REMOVED SLIMbus channel is inactive and removed

enum **slim_ch_data_fmt**

Constants

SLIM_CH_DATA_FMT_NOT_DEFINED Undefined

SLIM_CH_DATA_FMT_LPCM_AUDIO LPCM audio

SLIM_CH_DATA_FMT_IEC61937_COMP_AUDIO IEC61937 Compressed audio

SLIM_CH_DATA_FMT_PACKED_PDM_AUDIO Packed PDM audio

Description

Table 60 of SLIMbus Spec 1.01.01

enum **slim_ch_aux_bit_fmt**

Constants

SLIM_CH_AUX_FMT_NOT_APPLICABLE Undefined

SLIM_CH_AUX_FMT_ZCUV_TUNNEL_IEC60958 ZCUV for tunneling IEC60958

SLIM_CH_AUX_FMT_USER_DEFINED User defined

Description

Table 63 of SLIMbus Spec 2.0

struct **slim_channel**
SLIMbus channel, used for state machine

Definition

```
struct slim_channel {
  int id;
  int prrate;
  int seg_dist;
  enum slim_ch_data_fmt data_fmt;
  enum slim_ch_aux_bit_fmt aux_fmt;
  enum slim_channel_state state;
};
```

Members

id ID of channel

prrate Presense rate of channel from Table 66 of SLIMbus 2.0 Specs

seg_dist segment distribution code from Table 20 of SLIMbus 2.0 Specs

data_fmt Data format of channel.

aux_fmt Aux format for this channel.

state channel state machine

struct **slim_port**
SLIMbus port

Definition

```
struct slim_port {
  int id;
  enum slim_port_direction direction;
  enum slim_port_state state;
  struct slim_channel ch;
};
```

Members

id Port id

direction Port direction, Source or Sink.

state state machine of port.

ch channel associated with this port.

enum **slim_transport_protocol**

Constants

SLIM_PROTO_ISO Isochronous Protocol, no flow control as data rate match channel rate flow control embedded in the data.

SLIM_PROTO_PUSH Pushed Protocol, includes flow control, Used to carry data whose rate is equal to, or lower than the channel rate.

SLIM_PROTO_PULL Pulled Protocol, similar usage as pushed protocol but pull is a unicast.

SLIM_PROTO_LOCKED Locked Protocol

SLIM_PROTO_ASYNC_SMPLX Asynchronous Protocol-Simplex

SLIM_PROTO_ASYNC_HALF_DUP Asynchronous Protocol-Half-duplex

SLIM_PROTO_EXT_SMPLX Extended Asynchronous Protocol-Simplex

SLIM_PROTO_EXT_HALF_DUP Extended Asynchronous Protocol-Half-duplex

Description

Table 47 of SLIMbus 2.0 specs.

struct **slim_stream_runtime**
SLIMbus stream runtime instance

Definition

```
struct slim_stream_runtime {
  const char *name;
  struct slim_device *dev;
  int direction;
  enum slim_transport_protocol prot;
  unsigned int rate;
  unsigned int bps;
  unsigned int ratem;
  int num_ports;
  struct slim_port *ports;
```

(continues on next page)

(continued from previous page)

```
  struct list_head node;
};
```

Members

name Name of the stream

dev SLIM Device instance associated with this stream

direction direction of stream

prot Transport protocol used in this stream

rate Data rate of samples *

bps bits per sample

ratem rate multipler which is super frame rate/data rate

num_ports number of ports

ports pointer to instance of ports

node list head for stream associated with slim device.

struct **slim_controller**
Controls every instance of SLIMbus (similar to 'master' on SPI)

Definition

```
struct slim_controller {
  struct device           *dev;
  unsigned int            id;
  char name[SLIMBUS_NAME_SIZE];
  int min_cg;
  int max_cg;
  int clkgear;
  struct ida              laddr_ida;
  struct slim_framer      *a_framer;
  struct mutex            lock;
  struct list_head        devices;
  struct idr              tid_idr;
  spinlock_t txn_lock;
  struct slim_sched       sched;
  int (*xfer_msg)(struct slim_controller *ctrl, struct slim_msg_txn *tx);
  int (*set_laddr)(struct slim_controller *ctrl, struct slim_eaddr *ea, u8␣
↪laddr);
  int (*get_laddr)(struct slim_controller *ctrl, struct slim_eaddr *ea, u8␣
↪*laddr);
  int (*enable_stream)(struct slim_stream_runtime *rt);
  int (*disable_stream)(struct slim_stream_runtime *rt);
  int (*wakeup)(struct slim_controller *ctrl);
};
```

Members

dev Device interface to this driver

id Board-specific number identifier for this controller/bus

name Name for this controller

min_cg Minimum clock gear supported by this controller (default value: 1)

max_cg Maximum clock gear supported by this controller (default value: 10)

clkgear Current clock gear in which this bus is running

laddr_ida logical address id allocator

a_framer Active framer which is clocking the bus managed by this controller

lock Mutex protecting controller data structures

devices Slim device list

tid_idr tid id allocator

txn_lock Lock to protect table of transactions

sched scheduler structure used by the controller

xfer_msg Transfer a message on this controller (this can be a broadcast control/status message like data channel setup, or a unicast message like value element read/write.

set_laddr Setup logical address at laddr for the slave with elemental address e_addr. Drivers implementing controller will be expected to send unicast message to this device with its logical address.

get_laddr It is possible that controller needs to set fixed logical address table and get_laddr can be used in that case so that controller can do this assignment. Use case is when the master is on the remote processor side, who is resposible for allocating laddr.

enable_stream This function pointer implements controller-specific procedure to enable a stream.

disable_stream This function pointer implements controller-specific procedure to disable stream.

'Manager device' is responsible for device management, bandwidth allocation, channel setup, and port associations per channel. Device management means Logical address assignment/removal based on enumeration (report-present, report-absent) of a device. Bandwidth allocation is done dynamically by the manager based on active channels on the bus, message-bandwidth requests made by SLIMbus devices. Based on current bandwidth usage, manager chooses a frequency to run the bus at (in steps of 'clock-gear', 1 through 10, each clock gear representing twice the frequency than the previous gear). Manager is also responsible for entering (and exiting) low-power-mode (known as 'clock pause'). Manager can do handover of framer if there are multiple framers on the bus and a certain usecase warrants using certain framer to avoid keeping previous framer being powered-on.

Controller here performs duties of the manager device, and 'interface device'. Interface device is responsible for monitoring the bus and reporting information such as loss-of-synchronization, data slot-collision.

wakeup This function pointer implements controller-specific procedure to wake it up from clock-pause. Framework will call this to bring the controller out of clock pause.

int **slim_unregister_controller**(struct slim_controller * ctrl)
Controller tear-down.

Parameters

struct slim_controller * ctrl Controller to tear-down.

void **slim_report_absent**(struct slim_device * sbdev)
Controller calls this function when a device reports absent, OR when the device cannot be communicated with

Parameters

struct slim_device * sbdev Device that cannot be reached, or sent report absent

struct slim_device * **slim_get_device**(struct slim_controller * ctrl, struct slim_eaddr * e_addr)
get handle to a device.

Parameters

struct slim_controller * ctrl Controller on which this device will be added/queried

struct slim_eaddr * e_addr Enumeration address of the device to be queried

Return

pointer to a device if it has already reported. Creates a new device and returns pointer to it if the device has not yet enumerated.

struct slim_device * **of_slim_get_device**(struct slim_controller * ctrl, struct device_node * np)
get handle to a device using dt node.

Parameters

struct slim_controller * ctrl Controller on which this device will be added/queried

struct device_node * np node pointer to device

Return

pointer to a device if it has already reported. Creates a new device and returns pointer to it if the device has not yet enumerated.

int **slim_device_report_present**(struct slim_controller * ctrl, struct slim_eaddr * e_addr, u8 * laddr)
Report enumerated device.

Parameters

struct slim_controller * ctrl Controller with which device is enumerated.

struct slim_eaddr * e_addr Enumeration address of the device.

u8 * laddr Return logical address (if valid flag is false)

Description

Called by controller in response to REPORT_PRESENT. Framework will assign a logical address to this enumeration address. Function returns -EXFULL to indicate that all logical addresses are already taken.

int **slim_get_logical_addr**(struct slim_device * sbdev)
get/allocate logical address of a SLIMbus device.

Parameters

struct slim_device * sbdev client handle requesting the address.

Return

zero if a logical address is valid or a new logical address has been assigned. error code in case of error.

57.1.6 Clock-pause:

SLIMbus mandates that a reconfiguration sequence (known as clock-pause) be broadcast to all active devices on the bus before the bus can enter low-power mode. Controller uses this sequence when it decides to enter low-power mode so that corresponding clocks and/or power-rails can be turned off to save power. Clock-pause is exited by waking up framer device (if controller driver initiates exiting low power mode), or by toggling the data line (if a slave device wants to initiate it).

Clock-pause APIs:

int **slim_ctrl_clk_pause**(struct slim_controller * ctrl, bool wakeup, u8 restart)
Called by slimbus controller to enter/exit 'clock pause'

Parameters

struct slim_controller * ctrl controller requesting bus to be paused or woken up

bool wakeup Wakeup this controller from clock pause.

u8 restart Restart time value per spec used for clock pause. This value isn't used when controller is to be woken up.

Description

Slimbus specification needs this sequence to turn-off clocks for the bus. The sequence involves sending 3 broadcast messages (reconfiguration sequence) to inform all devices on the bus. To exit clock-pause, controller typically wakes up active framer device. This API executes clock pause reconfiguration sequence if wakeup is false. If wakeup is true, controller's wakeup is called. For entering clock-pause, -EBUSY is returned if a message txn in pending.

57.1.7 Messaging:

The framework supports regmap and read/write apis to exchange control-information with a SLIMbus device. APIs can be synchronous or asynchronous. The header file <linux/slimbus.h> has more documentation about messaging APIs.

Messaging APIs:

void **slim_msg_response**(struct slim_controller * ctrl, u8 * reply, u8 tid, u8 len)
Deliver Message response received from a device to the framework.

Parameters

struct slim_controller * ctrl Controller handle

u8 * reply Reply received from the device

u8 tid Transaction ID received with which framework can associate reply.

u8 len Length of the reply

Description

Called by controller to inform framework about the response received. This helps in making the API asynchronous, and controller-driver doesn't need to manage 1 more table other than the one managed by framework mapping TID with buffers

int **slim_alloc_txn_tid**(struct slim_controller * ctrl, struct slim_msg_txn * txn)
Allocate a tid to txn

Parameters

struct slim_controller * ctrl Controller handle

struct slim_msg_txn * txn transaction to be allocated with tid.

Return

zero on success with valid txn->tid and error code on failures.

void **slim_free_txn_tid**(struct slim_controller * ctrl, struct slim_msg_txn * txn)
Freee tid of txn

Parameters

struct slim_controller * ctrl Controller handle

struct slim_msg_txn * txn transaction whose tid should be freed

int **slim_do_transfer**(struct slim_controller * ctrl, struct slim_msg_txn * txn)
Process a SLIMbus-messaging transaction

Parameters

struct slim_controller * ctrl Controller handle

struct slim_msg_txn * txn Transaction to be sent over SLIMbus

Description

Called by controller to transmit messaging transactions not dealing with Interface/Value elements. (e.g. transmittting a message to assign logical address to a slave device

Return

-ETIMEDOUT: If transmission of this message timed out (e.g. due to bus lines not being clocked or driven by controller)

int **slim_xfer_msg**(struct slim_device * sbdev, struct slim_val_inf * msg, u8 mc)
Transfer a value info message on slim device

Parameters

struct slim_device * sbdev slim device to which this msg has to be transfered

struct slim_val_inf * msg value info message pointer

u8 mc message code of the message

Description

Called by drivers which want to transfer a vlaue or info elements.

Return

-ETIMEDOUT: If transmission of this message timed out

int **slim_read**(struct slim_device * sdev, u32 addr, size_t count, u8 * val)
Read SLIMbus value element

Parameters

struct slim_device * sdev client handle.

u32 addr address of value element to read.

size_t count number of bytes to read. Maximum bytes allowed are 16.

u8 * val will return what the value element value was

Return

-EINVAL for Invalid parameters, -ETIMEDOUT If transmission of this message timed out (e.g. due to bus lines not being clocked or driven by controller)

int **slim_readb**(struct slim_device * sdev, u32 addr)
Read byte from SLIMbus value element

Parameters

struct slim_device * sdev client handle.

u32 addr address in the value element to read.

Return

byte value of value element.

int **slim_write**(struct slim_device * sdev, u32 addr, size_t count, u8 * val)
Write SLIMbus value element

Parameters

struct slim_device * sdev client handle.

u32 addr address in the value element to write.

size_t count number of bytes to write. Maximum bytes allowed are 16.

u8 * val value to write to value element

Return

-EINVAL for Invalid parameters, -ETIMEDOUT If transmission of this message timed out (e.g. due to bus lines not being clocked or driven by controller)

int **slim_writeb**(struct slim_device * sdev, u32 addr, u8 value)
Write byte to SLIMbus value element

Parameters

struct slim_device * sdev client handle.

u32 addr address of value element to write.

u8 value value to write to value element

Return

-EINVAL for Invalid parameters, -ETIMEDOUT If transmission of this message timed out (e.g. due to bus lines not being clocked or driven by controller)

Streaming APIs:

struct slim_stream_runtime * **slim_stream_allocate**(struct slim_device * dev, const char * name)
Allocate a new SLIMbus Stream

Parameters

struct slim_device * dev Slim device to be associated with

const char * name name of the stream

Description

This is very first call for SLIMbus streaming, this API will allocate a new SLIMbus stream and return a valid stream runtime pointer for client to use it in subsequent stream apis. state of stream is set to ALLOCATED

Return

valid pointer on success and error code on failure. From ASoC DPCM framework, this state is linked to startup() operation.

int **slim_stream_prepare**(struct slim_stream_runtime * rt, struct slim_stream_config * cfg)
Prepare a SLIMbus Stream

Parameters

struct slim_stream_runtime * rt instance of slim stream runtime to configure

struct slim_stream_config * cfg new configuration for the stream

Description

This API will configure SLIMbus stream with config parameters from cfg. return zero on success and error code on failure. From ASoC DPCM framework, this state is linked to hw_params() operation.

int **slim_stream_enable**(struct slim_stream_runtime * stream)
Enable a prepared SLIMbus Stream

Parameters

struct slim_stream_runtime * stream instance of slim stream runtime to enable

Description

This API will enable all the ports and channels associated with SLIMbus stream

Return

zero on success and error code on failure. From ASoC DPCM framework, this state is linked to trigger() start operation.

int **slim_stream_disable**(struct slim_stream_runtime * stream)
Disable a SLIMbus Stream

Parameters

struct slim_stream_runtime * stream instance of slim stream runtime to disable

Description

This API will disable all the ports and channels associated with SLIMbus stream

Return

zero on success and error code on failure. From ASoC DPCM framework, this state is linked to trigger() pause operation.

int **slim_stream_unprepare**(struct slim_stream_runtime * stream)
Un-prepare a SLIMbus Stream

Parameters

struct slim_stream_runtime * stream instance of slim stream runtime to unprepare

Description

This API will un allocate all the ports and channels associated with SLIMbus stream

Return

zero on success and error code on failure. From ASoC DPCM framework, this state is linked to trigger() stop operation.

int **slim_stream_free**(struct slim_stream_runtime * stream)
Free a SLIMbus Stream

Parameters

struct slim_stream_runtime * stream instance of slim stream runtime to free

Description

This API will un allocate all the memory associated with slim stream runtime, user is not allowed to make an dereference to stream after this call.

Return

zero on success and error code on failure. From ASoC DPCM framework, this state is linked to shutdown() operation.

CHAPTER

FIFTYEIGHT

SOUNDWIRE DOCUMENTATION

58.1 SoundWire Subsystem Summary

SoundWire is a new interface ratified in 2015 by the MIPI Alliance. SoundWire is used for transporting data typically related to audio functions. SoundWire interface is optimized to integrate audio devices in mobile or mobile inspired systems.

SoundWire is a 2-pin multi-drop interface with data and clock line. It facilitates development of low cost, efficient, high performance systems. Broad level key features of SoundWire interface include:

(1) Transporting all of payload data channels, control information, and setup commands over a single two-pin interface.

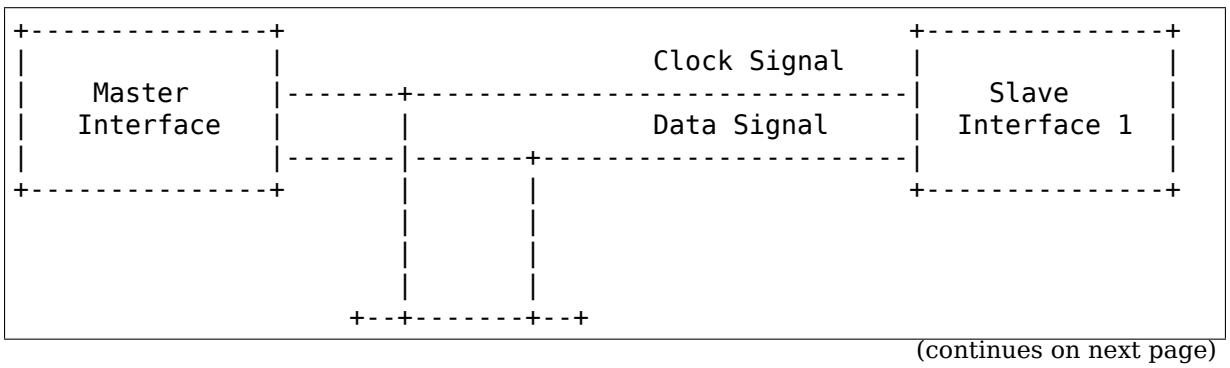
(2) Lower clock frequency, and hence lower power consumption, by use of DDR (Dual Data Rate) data transmission.

(3) Clock scaling and optional multiple data lanes to give wide flexibility in data rate to match system requirements.

(4) Device status monitoring, including interrupt-style alerts to the Master.

The SoundWire protocol supports up to eleven Slave interfaces. All the interfaces share the common Bus containing data and clock line. Each of the Slaves can support up to 14 Data Ports. 13 Data Ports are dedicated to audio transport. Data Port0 is dedicated to transport of Bulk control information, each of the audio Data Ports (1..14) can support up to 8 Channels in transmit or receiving mode (typically fixed direction but configurable direction is enabled by the specification). Bandwidth restrictions to ~19.2..24.576Mbits/s don't however allow for 11*13*8 channels to be transmitted simultaneously.

Below figure shows an example of connectivity between a SoundWire Master and two Slave devices.



(continues on next page)

(continued from previous page)

```
                    |             |
                    |   Slave     |
                    | Interface 2 |
                    |             |
                    +-------------+
```

58.1.1 Terminology

The MIPI SoundWire specification uses the term 'device' to refer to a Master or Slave interface, which of course can be confusing. In this summary and code we use the term interface only to refer to the hardware. We follow the Linux device model by mapping each Slave interface connected on the bus as a device managed by a specific driver. The Linux SoundWire subsystem provides a framework to implement a SoundWire Slave driver with an API allowing 3rd-party vendors to enable implementation-defined functionality while common setup/configuration tasks are handled by the bus.

Bus: Implements SoundWire Linux Bus which handles the SoundWire protocol. Programs all the MIPI-defined Slave registers. Represents a SoundWire Master. Multiple instances of Bus may be present in a system.

Slave: Registers as SoundWire Slave device (Linux Device). Multiple Slave devices can register to a Bus instance.

Slave driver: Driver controlling the Slave device. MIPI-specified registers are controlled directly by the Bus (and transmitted through the Master driver/interface). Any implementation-defined Slave register is controlled by Slave driver. In practice, it is expected that the Slave driver relies on regmap and does not request direct register access.

58.1.2 Programming interfaces (SoundWire Master interface Driver)

SoundWire Bus supports programming interfaces for the SoundWire Master implementation and SoundWire Slave devices. All the code uses the "sdw" prefix commonly used by SoC designers and 3rd party vendors.

Each of the SoundWire Master interfaces needs to be registered to the Bus. Bus implements API to read standard Master MIPI properties and also provides callback in Master ops for Master driver to implement its own functions that provides capabilities information. DT support is not implemented at this time but should be trivial to add since capabilities are enabled with the `device_property_` API.

The Master interface along with the Master interface capabilities are registered based on board file, DT or ACPI.

Following is the Bus API to register the SoundWire Bus:

```
int sdw_bus_master_add(struct sdw_bus *bus,
                       struct device *parent,
                       struct fwnode_handle)
```

(continues on next page)

(continued from previous page)

```
{
        sdw_master_device_add(bus, parent, fwnode);

        mutex_init(&bus->lock);
        INIT_LIST_HEAD(&bus->slaves);

        /* Check ACPI for Slave devices */
        sdw_acpi_find_slaves(bus);

        /* Check DT for Slave devices */
        sdw_of_find_slaves(bus);

        return 0;
}
```

This will initialize sdw_bus object for Master device. “sdw_master_ops” and “sdw_master_port_ops” callback functions are provided to the Bus.

“sdw_master_ops” is used by Bus to control the Bus in the hardware specific way. It includes Bus control functions such as sending the SoundWire read/write messages on Bus, setting up clock frequency & Stream Synchronization Point (SSP). The “sdw_master_ops”structure abstracts the hardware details of the Master from the Bus.

“sdw_master_port_ops” is used by Bus to setup the Port parameters of the Master interface Port. Master interface Port register map is not defined by MIPI specification, so Bus calls the “sdw_master_port_ops”callback function to do Port operations like “Port Prepare” , “Port Transport params set” , “Port enable and disable” . The implementation of the Master driver can then perform hardware-specific configurations.

58.1.3 Programming interfaces (SoundWire Slave Driver)

The MIPI specification requires each Slave interface to expose a unique 48-bit identifier, stored in 6 read-only dev_id registers. This dev_id identifier contains vendor and part information, as well as a field enabling to differentiate between identical components. An additional class field is currently unused. Slave driver is written for a specific vendor and part identifier, Bus enumerates the Slave device based on these two ids. Slave device and driver match is done based on these two ids . Probe of the Slave driver is called by Bus on successful match between device and driver id. A parent/child relationship is enforced between Master and Slave devices (the logical representation is aligned with the physical connectivity).

The information on Master/Slave dependencies is stored in platform data, boardfile, ACPI or DT. The MIPI Software specification defines additional link_id parameters for controllers that have multiple Master interfaces. The dev_id registers are only unique in the scope of a link, and the link_id unique in the scope of a controller. Both dev_id and link_id are not necessarily unique at the system level but the parent/child information is used to avoid ambiguity.

```
static const struct sdw_device_id slave_id[] = {
        SDW_SLAVE_ENTRY(0x025d, 0x700, 0),
```

(continues on next page)

(continued from previous page)

```
        {},
};
MODULE_DEVICE_TABLE(sdw, slave_id);

static struct sdw_driver slave_sdw_driver = {
        .driver = {
                   .name = "slave_xxx",
                   .pm = &slave_runtime_pm,
                   },
        .probe = slave_sdw_probe,
        .remove = slave_sdw_remove,
        .ops = &slave_slave_ops,
        .id_table = slave_id,
};
```

For capabilities, Bus implements API to read standard Slave MIPI properties and also provides callback in Slave ops for Slave driver to implement own function that provides capabilities information. Bus needs to know a set of Slave capabilities to program Slave registers and to control the Bus reconfigurations.

58.1.4 Future enhancements to be done

(1) Bulk Register Access (BRA) transfers.

(2) Multiple data lane support.

58.1.5 Links

SoundWire MIPI specification 1.1 is available at: https://members.mipi.org/wg/All-Members/document/70290

SoundWire MIPI DisCo (Discovery and Configuration) specification is available at: https://www.mipi.org/specifications/mipi-disco-soundwire

(publicly accessible with registration or directly accessible to MIPI members)

MIPI Alliance Manufacturer ID Page: mid.mipi.org

58.2 Audio Stream in SoundWire

An audio stream is a logical or virtual connection created between

(1) System memory buffer(s) and Codec(s)

(2) DSP memory buffer(s) and Codec(s)

(3) FIFO(s) and Codec(s)

(4) Codec(s) and Codec(s)

which is typically driven by a DMA(s) channel through the data link. An audio stream contains one or more channels of data. All channels within stream must have same sample rate and same sample size.

Assume a stream with two channels (Left & Right) is opened using SoundWire interface. Below are some ways a stream can be represented in SoundWire.

Stream Sample in memory (System memory, DSP memory or FIFOs)

```
------------------------
| L | R | L | R | L | R |
------------------------
```

Example 1: Stereo Stream with L and R channels is rendered from Master to Slave. Both Master and Slave is using single port.

```
+---------------+                    Clock Signal  +---------------+
|    Master     +-----------------------------------+     Slave     |
|   Interface   |                                   |   Interface   |
|               |                                   |       1       |
|               |                     Data Signal   |               |
|    L  +  R    +-----------------------------------+    L  +  R    |
|     (Data)    |     Data Direction                |     (Data)    |
+---------------+  +----------------------->       +---------------+
```

Example 2: Stereo Stream with L and R channels is captured from Slave to Master. Both Master and Slave is using single port.

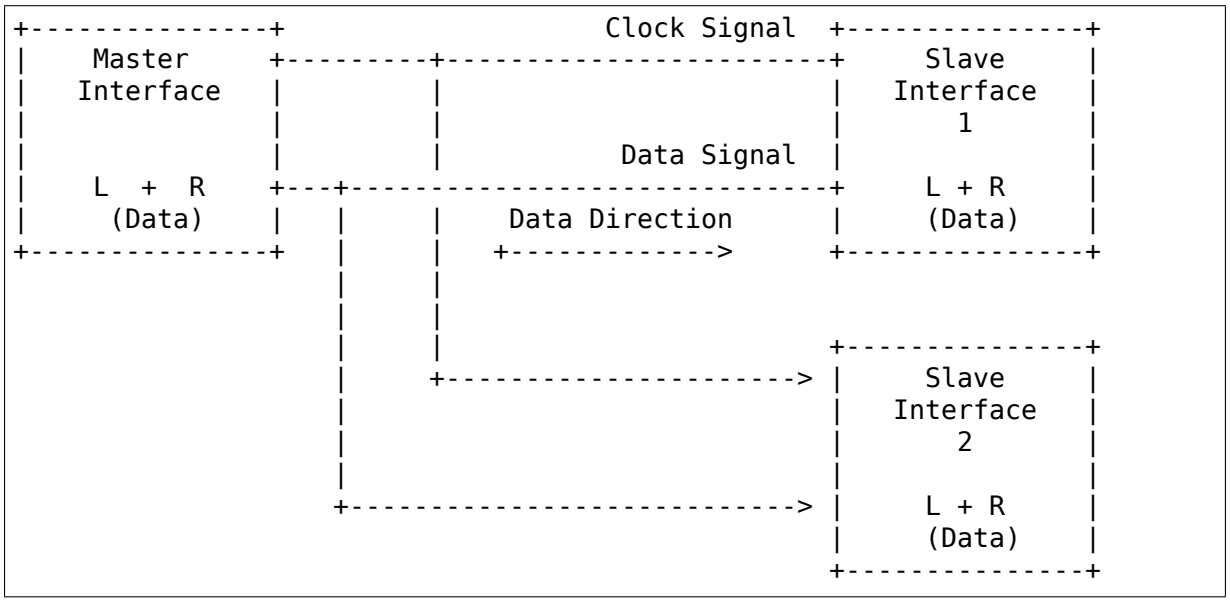
```
+---------------+                    Clock Signal  +---------------+
|    Master     +-----------------------------------+     Slave     |
|   Interface   |                                   |   Interface   |
|               |                                   |       1       |
|               |                     Data Signal   |               |
|    L  +  R    +-----------------------------------+    L  +  R    |
|     (Data)    |     Data Direction                |     (Data)    |
+---------------+  <-----------------------+       +---------------+
```

Example 3: Stereo Stream with L and R channels is rendered by Master. Each of the L and R channel is received by two different Slaves. Master and both Slaves are using single port.

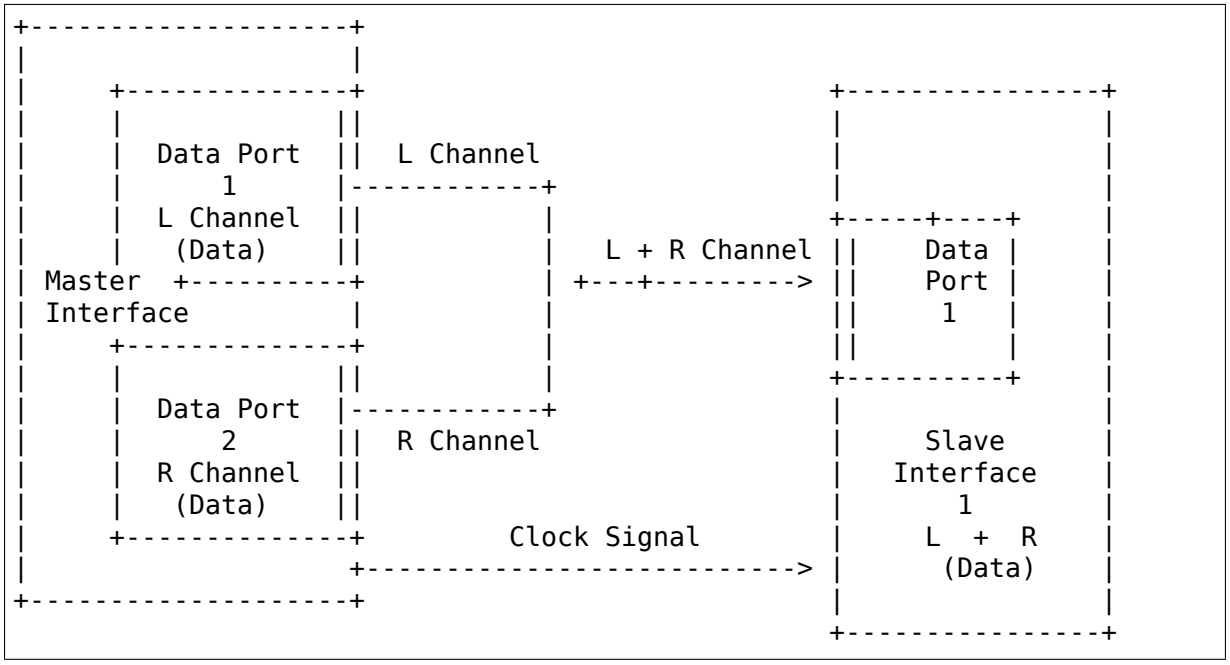
```
+---------------+                    Clock Signal  +---------------+
|    Master     +---------+-------------------------+     Slave     |
|   Interface   |         |                         |   Interface   |
|               |         |                         |       1       |
|               |         |           Data Signal   |               |
|    L  +  R    +---+-----------------------------+       L       |
|     (Data)    |   |     |    Data Direction       |     (Data)    |
+---------------+   |     |   +------------->       +---------------+
                    |     |
                    |     |
                    |     |                         +---------------+
                    |     +---------------------->  |     Slave     |
                    |                               |   Interface   |
                    |                               |       2       |
                    |                               |               |
                    +------------------------------>|       R       |
                                                    |     (Data)    |
                                                    +---------------+
```

Example 4: Stereo Stream with L and R channels is rendered by Master. Both

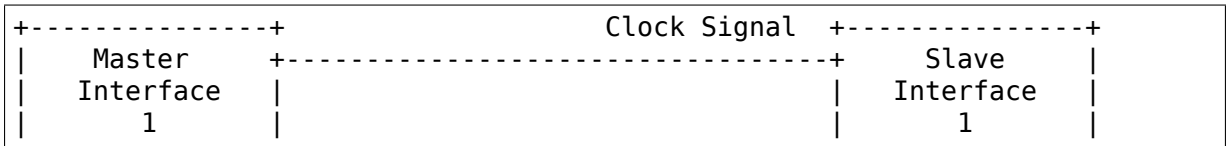
of the L and R channels are received by two different Slaves. Master and both Slaves are using single port handling L+R. Each Slave device processes the L + R data locally, typically based on static configuration or dynamic orientation, and may drive one or more speakers.



Example 5: Stereo Stream with L and R channel is rendered by two different Ports of the Master and is received by only single Port of the Slave interface.

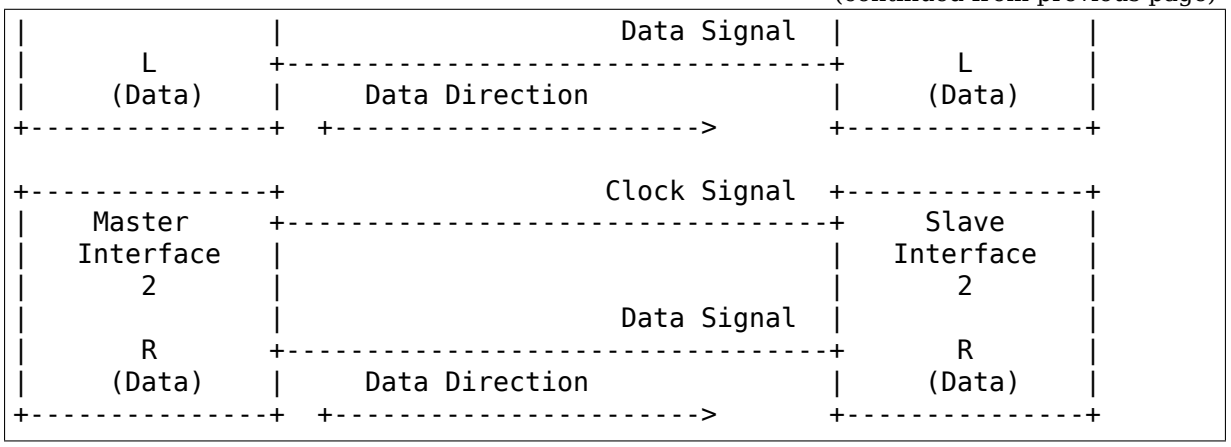


Example 6: Stereo Stream with L and R channel is rendered by 2 Masters, each rendering one channel, and is received by two different Slaves, each receiving one channel. Both Masters and both Slaves are using single port.

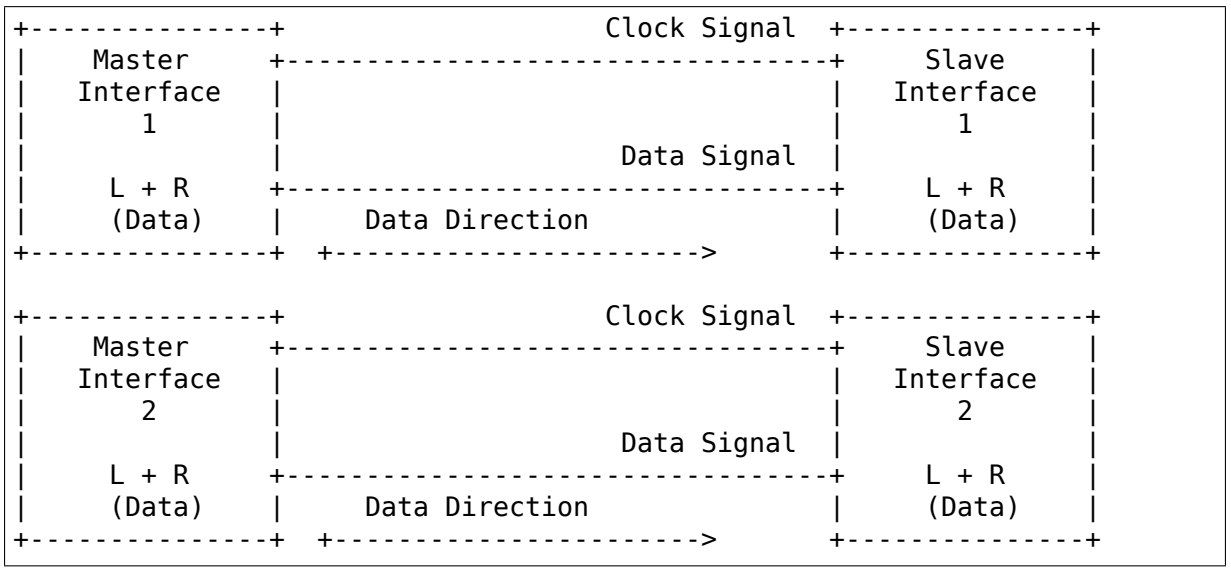


(continues on next page)

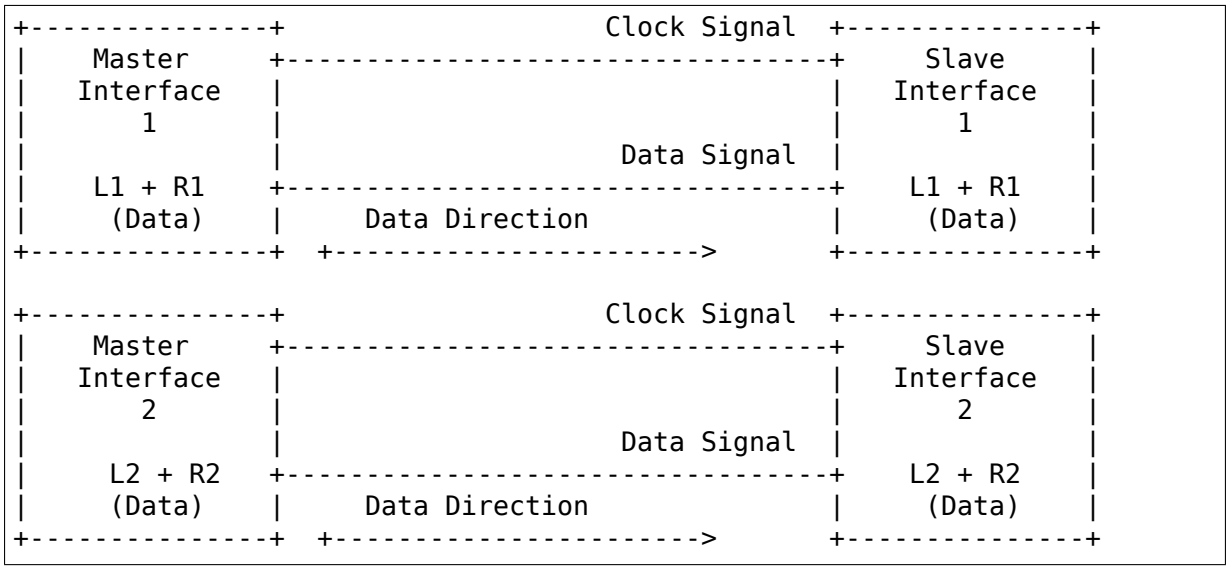
(continued from previous page)



Example 7: Stereo Stream with L and R channel is rendered by 2 Masters, each rendering both channels. Each Slave receives L + R. This is the same application as Example 4 but with Slaves placed on separate links.



Example 8: 4-channel Stream is rendered by 2 Masters, each rendering a 2 channels. Each Slave receives 2 channels.



Note1: In multi-link cases like above, to lock, one would acquire a global lock and then go on locking bus instances. But, in this case the caller framework(ASoC DPCM) guarantees that stream operations on a card are always serialized. So, there is no race condition and hence no need for global lock.

Note2: A Slave device may be configured to receive all channels transmitted on a link for a given Stream (Example 4) or just a subset of the data (Example 3). The configuration of the Slave device is not handled by a SoundWire subsystem API, but instead by the snd_soc_dai_set_tdm_slot() API. The platform or machine driver will typically configure which of the slots are used. For Example 4, the same slots would be used by all Devices, while for Example 3 the Slave Device1 would use e.g. Slot 0 and Slave device2 slot 1.

Note3: Multiple Sink ports can extract the same information for the same bitSlots in the SoundWire frame, however multiple Source ports shall be configured with different bitSlot configurations. This is the same limitation as with I2S/PCM TDM usages.

58.2.1 SoundWire Stream Management flow

Stream definitions

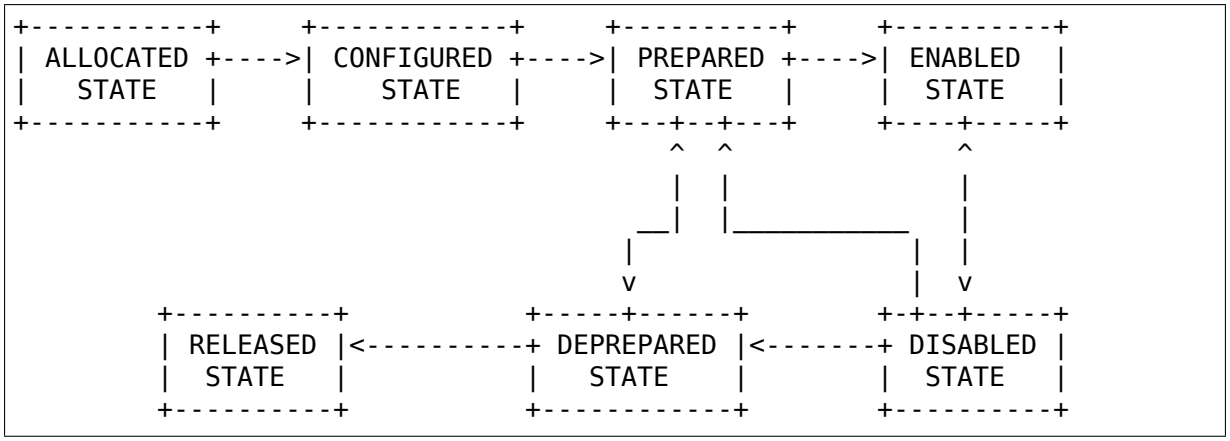
(1) Current stream: This is classified as the stream on which operation has to be performed like prepare, enable, disable, de-prepare etc.

(2) Active stream: This is classified as the stream which is already active on Bus other than current stream. There can be multiple active streams on the Bus.

SoundWire Bus manages stream operations for each stream getting rendered/captured on the SoundWire Bus. This section explains Bus operations done for each of the stream allocated/released on Bus. Following are the stream states maintained by the Bus for each of the audio stream.

SoundWire stream states

Below shows the SoundWire stream states and state transition diagram.



NOTE: State transitions between SDW_STREAM_ENABLED and SDW_STREAM_DISABLED are only relevant when then INFO_PAUSE flag is supported at the

ALSA/ASoC level. Likewise the transition between SDW_DISABLED_STATE and SDW_PREPARED_STATE depends on the INFO_RESUME flag.

NOTE2: The framework implements basic state transition checks, but does not e.g. check if a transition from DISABLED to ENABLED is valid on a specific platform. Such tests need to be added at the ALSA/ASoC level.

Stream State Operations

Below section explains the operations done by the Bus on Master(s) and Slave(s) as part of stream state transitions.

SDW_STREAM_ALLOCATED

Allocation state for stream. This is the entry state of the stream. Operations performed before entering in this state:

(1) A stream runtime is allocated for the stream. This stream runtime is used as a reference for all the operations performed on the stream.

(2) The resources required for holding stream runtime information are allocated and initialized. This holds all stream related information such as stream type (PCM/PDM) and parameters, Master and Slave interface associated with the stream, stream state etc.

After all above operations are successful, stream state is set to SDW_STREAM_ALLOCATED.

Bus implements below API for allocate a stream which needs to be called once per stream. From ASoC DPCM framework, this stream state maybe linked to .startup() operation.

```
int sdw_alloc_stream(char * stream_name);
```

SDW_STREAM_CONFIGURED

Configuration state of stream. Operations performed before entering in this state:

(1) The resources allocated for stream information in SDW_STREAM_ALLOCATED state are updated here. This includes stream parameters, Master(s) and Slave(s) runtime information associated with current stream.

(2) All the Master(s) and Slave(s) associated with current stream provide the port information to Bus which includes port numbers allocated by Master(s) and Slave(s) for current stream and their channel mask.

After all above operations are successful, stream state is set to SDW_STREAM_CONFIGURED.

Bus implements below APIs for CONFIG state which needs to be called by the respective Master(s) and Slave(s) associated with stream. These APIs can only be

invoked once by respective Master(s) and Slave(s). From ASoC DPCM framework, this stream state is linked to .hw_params() operation.

```
int sdw_stream_add_master(struct sdw_bus * bus,
              struct sdw_stream_config * stream_config,
              struct sdw_ports_config * ports_config,
              struct sdw_stream_runtime * stream);

int sdw_stream_add_slave(struct sdw_slave * slave,
              struct sdw_stream_config * stream_config,
              struct sdw_ports_config * ports_config,
              struct sdw_stream_runtime * stream);
```

SDW_STREAM_PREPARED

Prepare state of stream. Operations performed before entering in this state:

(0) Steps 1 and 2 are omitted in the case of a resume operation, where the bus bandwidth is known.

(1) Bus parameters such as bandwidth, frame shape, clock frequency, are computed based on current stream as well as already active stream(s) on Bus. Re-computation is required to accommodate current stream on the Bus.

(2) Transport and port parameters of all Master(s) and Slave(s) port(s) are computed for the current as well as already active stream based on frame shape and clock frequency computed in step 1.

(3) Computed Bus and transport parameters are programmed in Master(s) and Slave(s) registers. The banked registers programming is done on the alternate bank (bank currently unused). Port(s) are enabled for the already active stream(s) on the alternate bank (bank currently unused). This is done in order to not disrupt already active stream(s).

(4) Once all the values are programmed, Bus initiates switch to alternate bank where all new values programmed gets into effect.

(5) Ports of Master(s) and Slave(s) for current stream are prepared by programming PrepareCtrl register.

After all above operations are successful, stream state is set to `SDW_STREAM_PREPARED`.

Bus implements below API for PREPARE state which needs to be called once per stream. From ASoC DPCM framework, this stream state is linked to .prepare() operation. Since the .trigger() operations may not follow the .prepare(), a direct transition from `SDW_STREAM_PREPARED` to `SDW_STREAM_DEPREPARED` is allowed.

```
int sdw_prepare_stream(struct sdw_stream_runtime * stream);
```

SDW_STREAM_ENABLED

Enable state of stream. The data port(s) are enabled upon entering this state. Operations performed before entering in this state:

(1) All the values computed in SDW_STREAM_PREPARED state are programmed in alternate bank (bank currently unused). It includes programming of already active stream(s) as well.

(2) All the Master(s) and Slave(s) port(s) for the current stream are enabled on alternate bank (bank currently unused) by programming ChannelEn register.

(3) Once all the values are programmed, Bus initiates switch to alternate bank where all new values programmed gets into effect and port(s) associated with current stream are enabled.

After all above operations are successful, stream state is set to `SDW_STREAM_ENABLED`.

Bus implements below API for ENABLE state which needs to be called once per stream. From ASoC DPCM framework, this stream state is linked to .trigger() start operation.

```
int sdw_enable_stream(struct sdw_stream_runtime * stream);
```

SDW_STREAM_DISABLED

Disable state of stream. The data port(s) are disabled upon exiting this state. Operations performed before entering in this state:

(1) All the Master(s) and Slave(s) port(s) for the current stream are disabled on alternate bank (bank currently unused) by programming ChannelEn register.

(2) All the current configuration of Bus and active stream(s) are programmed into alternate bank (bank currently unused).

(3) Once all the values are programmed, Bus initiates switch to alternate bank where all new values programmed gets into effect and port(s) associated with current stream are disabled.

After all above operations are successful, stream state is set to `SDW_STREAM_DISABLED`.

Bus implements below API for DISABLED state which needs to be called once per stream. From ASoC DPCM framework, this stream state is linked to .trigger() stop operation.

When the INFO_PAUSE flag is supported, a direct transition to `SDW_STREAM_ENABLED` is allowed.

For resume operations where ASoC will use the .prepare() callback, the stream can transition from `SDW_STREAM_DISABLED` to `SDW_STREAM_PREPARED`, with all required settings restored but without updating the bandwidth and bit allocation.

```
int sdw_disable_stream(struct sdw_stream_runtime * stream);
```

SDW_STREAM_DEPREPARED

De-prepare state of stream. Operations performed before entering in this state:

(1) All the port(s) of Master(s) and Slave(s) for current stream are de-prepared by programming PrepareCtrl register.

(2) The payload bandwidth of current stream is reduced from the total bandwidth requirement of bus and new parameters calculated and applied by performing bank switch etc.

After all above operations are successful, stream state is set to `SDW_STREAM_DEPREPARED`.

Bus implements below API for DEPREPARED state which needs to be called once per stream. ALSA/ASoC do not have a concept of ‘deprepare’, and the mapping from this stream state to ALSA/ASoC operation may be implementation specific.

When the INFO_PAUSE flag is supported, the stream state is linked to the .hw_free() operation - the stream is not deprepared on a TRIGGER_STOP.

Other implementations may transition to the `SDW_STREAM_DEPREPARED` state on TRIGGER_STOP, should they require a transition through the `SDW_STREAM_PREPARED` state.

```
int sdw_deprepare_stream(struct sdw_stream_runtime * stream);
```

SDW_STREAM_RELEASED

Release state of stream. Operations performed before entering in this state:

(1) Release port resources for all Master(s) and Slave(s) port(s) associated with current stream.

(2) Release Master(s) and Slave(s) runtime resources associated with current stream.

(3) Release stream runtime resources associated with current stream.

After all above operations are successful, stream state is set to `SDW_STREAM_RELEASED`.

Bus implements below APIs for RELEASE state which needs to be called by all the Master(s) and Slave(s) associated with stream. From ASoC DPCM framework, this stream state is linked to .hw_free() operation.

```
int sdw_stream_remove_master(struct sdw_bus * bus,
             struct sdw_stream_runtime * stream);
int sdw_stream_remove_slave(struct sdw_slave * slave,
             struct sdw_stream_runtime * stream);
```

The .shutdown() ASoC DPCM operation calls below Bus API to release stream assigned as part of ALLOCATED state.

In .shutdown() the data structure maintaining stream state are freed up.

```
void sdw_release_stream(struct sdw_stream_runtime * stream);
```

58.2.2 Not Supported

1. A single port with multiple channels supported cannot be used between two streams or across stream. For example a port with 4 channels cannot be used to handle 2 independent stereo streams even though it' s possible in theory in SoundWire.

58.3 SoundWire Error Handling

The SoundWire PHY was designed with care and errors on the bus are going to be very unlikely, and if they happen it should be limited to single bit errors. Examples of this design can be found in the synchronization mechanism (sync loss after two errors) and short CRCs used for the Bulk Register Access.

The errors can be detected with multiple mechanisms:

1. Bus clash or parity errors: This mechanism relies on low-level detectors that are independent of the payload and usages, and they cover both control and audio data. The current implementation only logs such errors. Improvements could be invalidating an entire programming sequence and restarting from a known position. In the case of such errors outside of a control/command sequence, there is no concealment or recovery for audio data enabled by the SoundWire protocol, the location of the error will also impact its audibility (most-significant bits will be more impacted in PCM), and after a number of such errors are detected the bus might be reset. Note that bus clashes due to programming errors (two streams using the same bit slots) or electrical issues during the transmit/receive transition cannot be distinguished, although a recurring bus clash when audio is enabled is a indication of a bus allocation issue. The interrupt mechanism can also help identify Slaves which detected a Bus Clash or a Parity Error, but they may not be responsible for the errors so resetting them individually is not a viable recovery strategy.

2. Command status: Each command is associated with a status, which only covers transmission of the data between devices. The ACK status indicates that the command was received and will be executed by the end of the current frame. A NAK indicates that the command was in error and will not be applied. In case of a bad programming (command sent to non-existent Slave or to a non-implemented register) or electrical issue, no response signals the command was ignored. Some Master implementations allow for a command to be retransmitted several times. If the retransmission fails, backtracking and restarting the entire programming sequence might be a solution. Alternatively some implementations might directly issue a bus reset and re-enumerate all devices.

3. Timeouts: In a number of cases such as ChannelPrepare or ClockStopPrepare, the bus driver is supposed to poll a register field until it transitions to a NotFinished value of zero. The MIPI SoundWire spec 1.1 does not define timeouts but the MIPI SoundWire DisCo document adds recommendation on

timeouts. If such configurations do not complete, the driver will return a -ETIMEOUT. Such timeouts are symptoms of a faulty Slave device and are likely impossible to recover from.

Errors during global reconfiguration sequences are extremely difficult to handle:

1. BankSwitch: An error during the last command issuing a BankSwitch is difficult to backtrack from. Retransmitting the Bank Switch command may be possible in a single segment setup, but this can lead to synchronization problems when enabling multiple bus segments (a command with side effects such as frame reconfiguration would be handled at different times). A global hard-reset might be the best solution.

Note that SoundWire does not provide a mechanism to detect illegal values written in valid registers. In a number of cases the standard even mentions that the Slave might behave in implementation-defined ways. The bus implementation does not provide a recovery mechanism for such errors, Slave or Master driver implementers are responsible for writing valid values in valid registers and implement additional range checking if needed.

58.4 SoundWire Locking

This document explains locking mechanism of the SoundWire Bus. Bus uses following locks in order to avoid race conditions in Bus operations on shared resources.

- Bus lock
- Message lock

58.4.1 Bus lock

SoundWire Bus lock is a mutex and is part of Bus data structure (sdw_bus) which is used for every Bus instance. This lock is used to serialize each of the following operations(s) within SoundWire Bus instance.

- Addition and removal of Slave(s), changing Slave status.
- Prepare, Enable, Disable and De-prepare stream operations.
- Access of Stream data structure.

58.4.2 Message lock

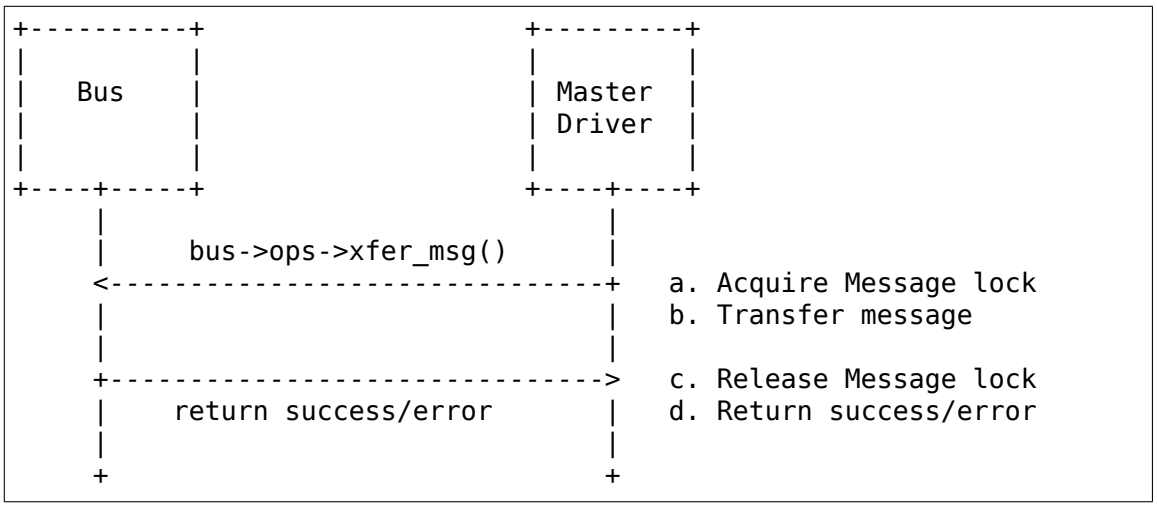
SoundWire message transfer lock. This mutex is part of Bus data structure (sdw_bus). This lock is used to serialize the message transfers (read/write) within a SoundWire Bus instance.

Below examples show how locks are acquired.

Example 1

Message transfer.

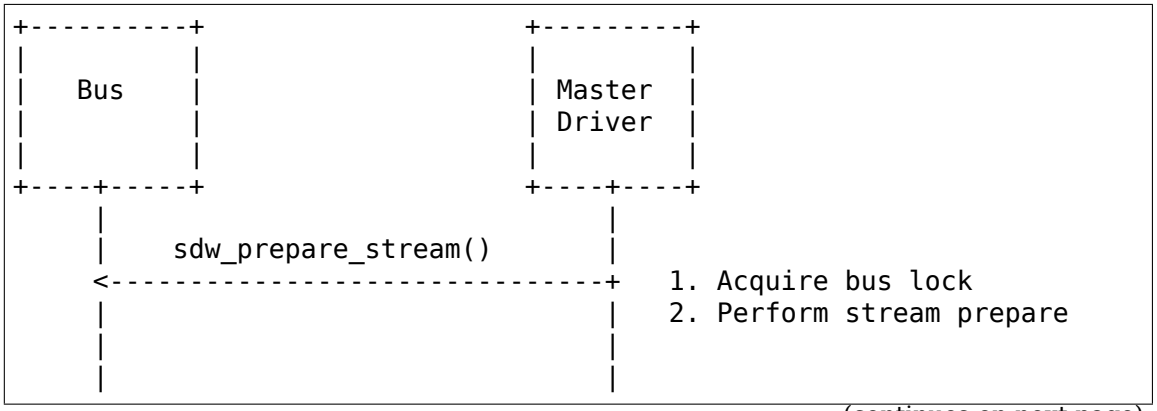
1. For every message transfer
   a. Acquire Message lock.
   b. Transfer message (Read/Write) to Slave1 or broadcast message on Bus in case of bank switch.
   c. Release Message lock



Example 2

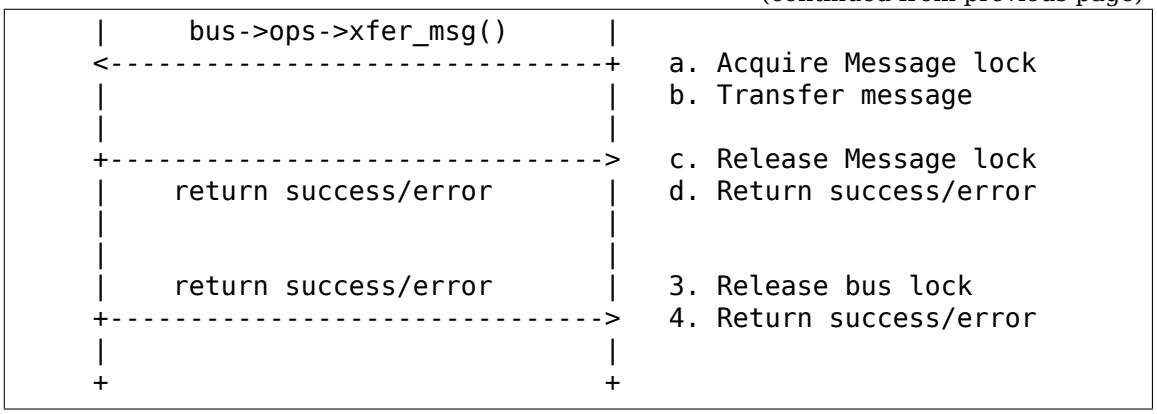
Prepare operation.

1. Acquire lock for Bus instance associated with Master 1.
2. For every message transfer in Prepare operation
   a. Acquire Message lock.
   b. Transfer message (Read/Write) to Slave1 or broadcast message on Bus in case of bank switch.
   c. Release Message lock.
3. Release lock for Bus instance associated with Master 1



(continues on next page)

(continued from previous page)



CHAPTER

FIFTYNINE

THERMAL

59.1 CPU cooling APIs How To

Written by Amit Daniel Kachhap <amit.kachhap@linaro.org>

Updated: 6 Jan 2015

59.1.1 0. Introduction

The generic cpu cooling(freq clipping) provides registration/unregistration APIs to the caller. The binding of the cooling devices to the trip point is left for the user. The registration APIs returns the cooling device pointer.

59.1.2 1. cpu cooling APIs

1.1 cpufreq registration/unregistration APIs

```
struct thermal_cooling_device
*cpufreq_cooling_register(struct cpumask *clip_cpus)
```

This interface function registers the cpufreq cooling device with the name “thermal-cpufreq-%x” . This api can support multiple instances of cpufreq cooling devices.

clip_cpus:

cpumask of cpus where the frequency constraints will happen.

```
struct thermal_cooling_device
*of_cpufreq_cooling_register(struct cpufreq_policy *policy)
```

This interface function registers the cpufreq cooling device with the name “thermal-cpufreq-%x” linking it with a device tree node, in order to bind it via the thermal DT code. This api can support multiple instances of cpufreq cooling devices.

policy: CPUFreq policy.

```
void cpufreq_cooling_unregister(struct thermal_cooling_device␣
↪*cdev)
```

This interface function unregisters the “thermal-cpufreq-%x”cooling device.

cdev: Cooling device pointer which has to be unregistered.

59.1.3 2. Power models

The power API registration functions provide a simple power model for CPUs. The current power is calculated as dynamic power (static power isn’t supported currently). This power model requires that the operating-points of the CPUs are registered using the kernel’s opp library and the cpufreq_frequency_table is assigned to the struct device of the cpu. If you are using CONFIG_CPUFREQ_DT then the cpufreq_frequency_table should already be assigned to the cpu device.

The dynamic power consumption of a processor depends on many factors. For a given processor implementation the primary factors are:

- The time the processor spends running, consuming dynamic power, as compared to the time in idle states where dynamic consumption is negligible. Herein we refer to this as ‘utilisation’.
- The voltage and frequency levels as a result of DVFS. The DVFS level is a dominant factor governing power consumption.
- In running time the ‘execution’ behaviour (instruction types, memory access patterns and so forth) causes, in most cases, a second order variation. In pathological cases this variation can be significant, but typically it is of a much lesser impact than the factors above.

A high level dynamic power consumption model may then be represented as:

```
Pdyn = f(run) * Voltage^2 * Frequency * Utilisation
```

f(run) here represents the described execution behaviour and its result has a units of Watts/Hz/Volt^2 (this often expressed in mW/MHz/uVolt^2)

The detailed behaviour for f(run) could be modelled on-line. However, in practice, such an on-line model has dependencies on a number of implementation specific processor support and characterisation factors. Therefore, in initial implementation that contribution is represented as a constant coefficient. This is a simplification consistent with the relative contribution to overall power variation.

In this simplified representation our model becomes:

```
Pdyn = Capacitance * Voltage^2 * Frequency * Utilisation
```

Where capacitance is a constant that represents an indicative running time dynamic power coefficient in fundamental units of mW/MHz/uVolt^2. Typical values for mobile CPUs might lie in range from 100 to 500. For reference, the approximate values for the SoC in ARM’s Juno Development Platform are 530 for the Cortex-A57 cluster and 140 for the Cortex-A53 cluster.

59.2 CPU Idle Cooling

59.2.1 Situation:

Under certain circumstances a SoC can reach a critical temperature limit and is unable to stabilize the temperature around a temperature control. When the SoC has to stabilize the temperature, the kernel can act on a cooling device to mitigate the dissipated power. When the critical temperature is reached, a decision must be taken to reduce the temperature, that, in turn impacts performance.

Another situation is when the silicon temperature continues to increase even after the dynamic leakage is reduced to its minimum by clock gating the component. This runaway phenomenon can continue due to the static leakage. The only solution is to power down the component, thus dropping the dynamic and static leakage that will allow the component to cool down.

Last but not least, the system can ask for a specific power budget but because of the OPP density, we can only choose an OPP with a power budget lower than the requested one and under-utilize the CPU, thus losing performance. In other words, one OPP under-utilizes the CPU with a power less than the requested power budget and the next OPP exceeds the power budget. An intermediate OPP could have been used if it were present.

59.2.2 Solutions:

If we can remove the static and the dynamic leakage for a specific duration in a controlled period, the SoC temperature will decrease. Acting on the idle state duration or the idle cycle injection period, we can mitigate the temperature by modulating the power budget.

The Operating Performance Point (OPP) density has a great influence on the control precision of cpufreq, however different vendors have a plethora of OPP density, and some have large power gap between OPPs, that will result in loss of performance during thermal control and loss of power in other scenarios.

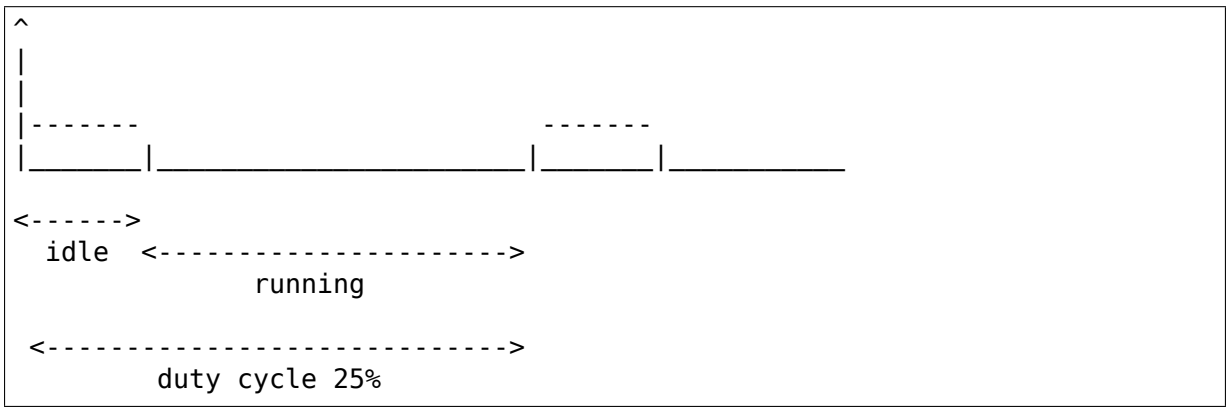
At a specific OPP, we can assume that injecting idle cycle on all CPUs belong to the same cluster, with a duration greater than the cluster idle state target residency, we lead to dropping the static and the dynamic leakage for this period (modulo the energy needed to enter this state). So the sustainable power with idle cycles has a linear relation with the OPP’ s sustainable power and can be computed with a coefficient similar to:

Power(IdleCycle) = Coef x Power(OPP)

59.2.3 Idle Injection:

The base concept of the idle injection is to force the CPU to go to an idle state for a specified time each control cycle, it provides another way to control CPU power and heat in addition to cpufreq. Ideally, if all CPUs belonging to the same cluster, inject their idle cycles synchronously, the cluster can reach its power down state with a minimum power consumption and reduce the static leakage to almost zero. However, these idle cycles injection will add extra latencies as the CPUs will have to wakeup from a deep sleep state.

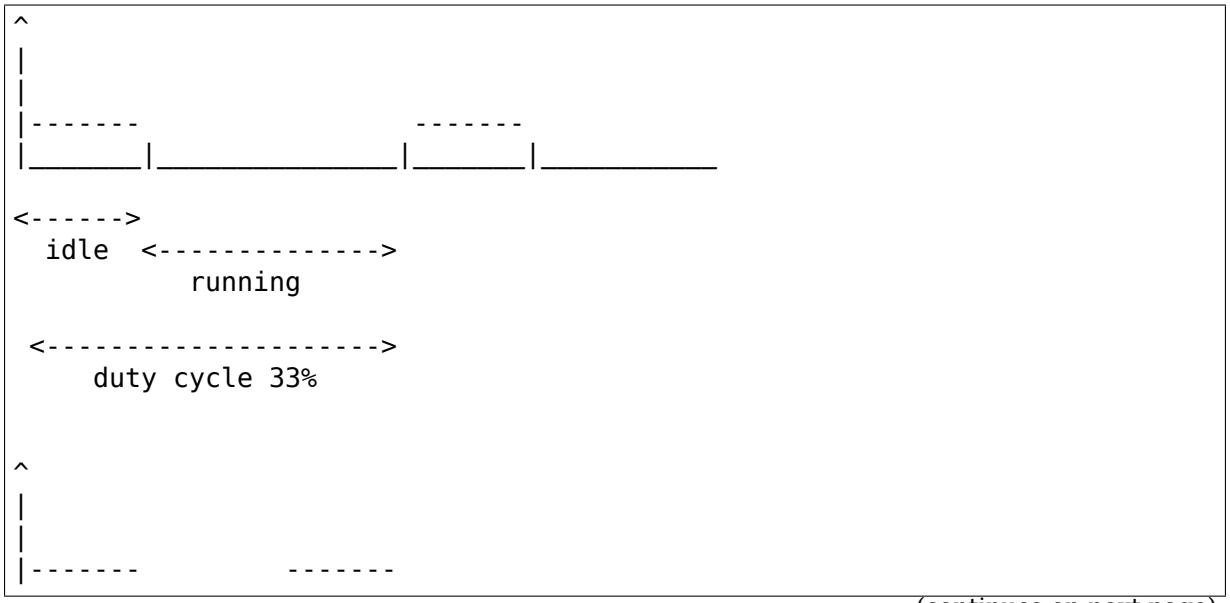
We use a fixed duration of idle injection that gives an acceptable performance penalty and a fixed latency. Mitigation can be increased or decreased by modulating the duty cycle of the idle injection.



The implementation of the cooling device bases the number of states on the duty cycle percentage. When no mitigation is happening the cooling device state is zero, meaning the duty cycle is 0%.

When the mitigation begins, depending on the governor's policy, a starting state is selected. With a fixed idle duration and the duty cycle (aka the cooling device state), the running duration can be computed.

The governor will change the cooling device state thus the duty cycle and this variation will modulate the cooling effect.



(continues on next page)

(continued from previous page)

```
|_______|_______|_______|____________

<------>
  idle  <------>
         running

 <------------->
  duty cycle 50%
```

The idle injection duration value must comply with the constraints:

- It is less than or equal to the latency we tolerate when the mitigation begins. It is platform dependent and will depend on the user experience, reactivity vs performance trade off we want. This value should be specified.
- It is greater than the idle state’s target residency we want to go for thermal mitigation, otherwise we end up consuming more energy.

59.2.4 Power considerations

When we reach the thermal trip point, we have to sustain a specified power for a specific temperature but at this time we consume:

Power = Capacitance x Voltage^2 x Frequency x Utilisation

…which is more than the sustainable power (or there is something wrong in the system setup). The ‘Capacitance’ and ‘Utilisation’ are a fixed value, ‘Voltage’ and the ‘Frequency’ are fixed artificially because we don’t want to change the OPP. We can group the ‘Capacitance’ and the ‘Utilisation’ into a single term which is the ‘Dynamic Power Coefficient (Cdyn)’ Simplifying the above, we have:

Pdyn = Cdyn x Voltage^2 x Frequency

The power allocator governor will ask us somehow to reduce our power in order to target the sustainable power defined in the device tree. So with the idle injection mechanism, we want an average power (Ptarget) resulting in an amount of time running at full power on a specific OPP and idle another amount of time. That could be put in a equation:

P(opp)target = ((Trunning x (P(opp)running) + (Tidle x P(opp)idle)) /

(Trunning + Tidle)

…

Tidle = Trunning x ((P(opp)running / P(opp)target) - 1)

At this point if we know the running period for the CPU, that gives us the idle injection we need. Alternatively if we have the idle injection duration, we can compute the running duration with:

Trunning = Tidle / ((P(opp)running / P(opp)target) - 1)

Practically, if the running power is less than the targeted power, we end up with a negative time value, so obviously the equation usage is bound to a power reduction,

hence a higher OPP is needed to have the running power greater than the targeted power.

However, in this demonstration we ignore three aspects:

- The static leakage is not defined here, we can introduce it in the equation but assuming it will be zero most of the time as it is difficult to get the values from the SoC vendors
- The idle state wake up latency (or entry + exit latency) is not taken into account, it must be added in the equation in order to rigorously compute the idle injection
- The injected idle duration must be greater than the idle state target residency, otherwise we end up consuming more energy and potentially invert the mitigation effect

So the final equation is:

Trunning = (Tidle - Twakeup) x (((P(opp)dyn + P(opp)static) - P(opp)target) / P(opp)target)

59.3 Generic Thermal Sysfs driver How To

Written by Sujith Thomas <sujith.thomas@intel.com>, Zhang Rui <rui.zhang@intel.com>

Updated: 2 January 2008

59.3.1 0. Introduction

The generic thermal sysfs provides a set of interfaces for thermal zone devices (sensors) and thermal cooling devices (fan, processor…) to register with the thermal management solution and to be a part of it.

This how-to focuses on enabling new thermal zone and cooling devices to participate in thermal management. This solution is platform independent and any type of thermal zone devices and cooling devices should be able to make use of the infrastructure.

The main task of the thermal sysfs driver is to expose thermal zone attributes as well as cooling device attributes to the user space. An intelligent thermal management application can make decisions based on inputs from thermal zone attributes (the current temperature and trip point temperature) and throttle appropriate devices.

- [0-*] denotes any positive number starting from 0
- [1-*] denotes any positive number starting from 1

59.3.2 1. thermal sysfs driver interface functions

1.1 thermal zone device interface

```
struct thermal_zone_device
*thermal_zone_device_register(char *type,
                              int trips, int mask, void *devdata,
                              struct thermal_zone_device_ops *ops,
                              const struct thermal_zone_params␣
↪*tzp,
                              int passive_delay, int polling_
↪delay))
```

This interface function adds a new thermal zone device (sensor) to /sys/class/thermal folder as thermal_zone[0-*]. It tries to bind all the thermal cooling devices registered at the same time.

type: the thermal zone type.

trips: the total number of trip points this thermal zone supports.

mask: Bit string: If 'n' th bit is set, then trip point 'n' is writeable.

devdata: device private data

ops: thermal zone device call-backs.

.bind: bind the thermal zone device with a thermal cooling device.

.unbind: unbind the thermal zone device with a thermal cooling device.

.get_temp: get the current temperature of the thermal zone.

.set_trips: set the trip points window. Whenever the current temperature is updated, the trip points immediately below and above the current temperature are found.

.get_mode: get the current mode (enabled/disabled) of the thermal zone.

- "enabled" means the kernel thermal management is enabled.
- "disabled" will prevent kernel thermal driver action upon trip points so that user applications can take charge of thermal management.

.set_mode: set the mode (enabled/disabled) of the thermal zone.

.get_trip_type: get the type of certain trip point.

.get_trip_temp: get the temperature above which the certain trip point will be fired.

.set_emul_temp: set the emulation temperature which helps in debugging different threshold temperature points.

tzp: thermal zone platform parameters.

passive_delay: number of milliseconds to wait between polls when performing passive cooling.

polling_delay: number of milliseconds to wait between polls when checking whether trip points have been crossed (0 for interrupt driven systems).

```
void thermal_zone_device_unregister(struct thermal_zone_device␣
↪*tz)
```

This interface function removes the thermal zone device. It deletes the corresponding entry from /sys/class/thermal folder and unbinds all the thermal cooling devices it uses.

```
struct thermal_zone_device
*thermal_zone_of_sensor_register(struct device *dev, int␣
↪sensor_id,
                      void *data,
                      const struct thermal_zone_of_device_
↪ops *ops)
```

This interface adds a new sensor to a DT thermal zone. This function will search the list of thermal zones described in device tree and look for the zone that refer to the sensor device pointed by dev->of_node as temperature providers. For the zone pointing to the sensor node, the sensor will be added to the DT thermal zone device.

The parameters for this interface are:

dev: Device node of sensor containing valid node pointer in dev->of_node.

sensor_id: a sensor identifier, in case the sensor IP has more than one sensors

data: a private pointer (owned by the caller) that will be passed back, when a temperature reading is needed.

ops: struct thermal_zone_of_device_ops *.

get_temp	a pointer to a function that reads the sensor temperature. This is mandatory callback provided by sensor driver.
set_trips	a pointer to a function that sets a temperature window. When this window is left the driver must inform the thermal core via thermal_zone_device_update.
get_trend	a pointer to a function that reads the sensor temperature trend.
set_emul_temp	a pointer to a function that sets sensor emulated temperature.

The thermal zone temperature is provided by the get_temp() function pointer of thermal_zone_of_device_ops. When called, it will have the private pointer @data back.

It returns error pointer if fails otherwise valid thermal zone

device handle. Caller should check the return handle with IS_ERR() for finding whether success or not.

```
void thermal_zone_of_sensor_unregister(struct device *dev,
                                       struct thermal_
↪zone_device *tzd)
```

This interface unregisters a sensor from a DT thermal zone which was successfully added by interface thermal_zone_of_sensor_register(). This function removes the sensor callbacks and private data from the thermal zone device registered with thermal_zone_of_sensor_register() interface. It will also silent the zone by remove the .get_temp() and get_trend() thermal zone device callbacks.

```
struct thermal_zone_device
*devm_thermal_zone_of_sensor_register(struct device *dev,
                      int sensor_id,
                      void *data,
                      const struct thermal_zone_of_device_
↪ops *ops)
```

This interface is resource managed version of thermal_zone_of_sensor_register().

All details of thermal_zone_of_sensor_register() described in section 1.1.3 is applicable here.

The benefit of using this interface to register sensor is that it is not require to explicitly call thermal_zone_of_sensor_unregister() in error path or during driver unbinding as this is done by driver resource manager.

```
void devm_thermal_zone_of_sensor_unregister(struct device␣
↪*dev,
                                    struct thermal_zone_
↪device *tzd)
```

This interface is resource managed version of thermal_zone_of_sensor_unregister(). All details of thermal_zone_of_sensor_unregister() described in section 1.1.4 is applicable here. Normally this function will not need to be called and the resource management code will ensure that the resource is freed.

```
int thermal_zone_get_slope(struct thermal_zone_device *tz)
```

This interface is used to read the slope attribute value for the thermal zone device, which might be useful for platform drivers for temperature calculations.

```
int thermal_zone_get_offset(struct thermal_zone_device␣
↪*tz)
```

This interface is used to read the offset attribute value for the

thermal zone device, which might be useful for platform drivers for temperature calculations.

1.2 thermal cooling device interface

```
struct thermal_cooling_device
*thermal_cooling_device_register(char *name,
                void *devdata, struct thermal_cooling_device_ops␣
↪*)
```

This interface function adds a new thermal cooling device (fan/processor/…) to /sys/class/thermal/ folder as cooling_device[0-*]. It tries to bind itself to all the thermal zone devices registered at the same time.

name: the cooling device name.

devdata: device private data.

ops: thermal cooling devices call-backs.

.get_max_state: get the Maximum throttle state of the cooling device.

.get_cur_state: get the Currently requested throttle state of the cooling device.

.set_cur_state: set the Current throttle state of the cooling device.

```
void thermal_cooling_device_unregister(struct thermal_cooling_
↪device *cdev)
```

This interface function removes the thermal cooling device. It deletes the corresponding entry from /sys/class/thermal folder and unbinds itself from all the thermal zone devices using it.

1.3 interface for binding a thermal zone device with a thermal cooling device

```
int thermal_zone_bind_cooling_device(struct thermal_zone_device␣
↪*tz,
        int trip, struct thermal_cooling_device *cdev,
        unsigned long upper, unsigned long lower, unsigned int␣
↪weight);
```

This interface function binds a thermal cooling device to a particular trip point of a thermal zone device.

This function is usually called in the thermal zone device .bind callback.

tz: the thermal zone device

cdev: thermal cooling device

trip: indicates which trip point in this thermal zone the cooling device is associated with.

upper: the Maximum cooling state for this trip point. THERMAL_NO_LIMIT means no upper limit, and the cooling device can be in max_state.

lower: the Minimum cooling state can be used for this trip point. THERMAL_NO_LIMIT means no lower limit, and the cooling device can be in cooling state 0.

weight: the influence of this cooling device in this thermal zone. See 1.4.1 below for more information.

```
int thermal_zone_unbind_cooling_device(struct thermal_zone_device␣
↪*tz,
                       int trip, struct thermal_cooling_device␣
↪*cdev);
```

This interface function unbinds a thermal cooling device from a particular trip point of a thermal zone device. This function is usually called in the thermal zone device .unbind callback.

tz: the thermal zone device

cdev: thermal cooling device

trip: indicates which trip point in this thermal zone the cooling device is associated with.

1.4 Thermal Zone Parameters

```
struct thermal_bind_params
```

This structure defines the following parameters that are used to bind a zone with a cooling device for a particular trip point.

.cdev: The cooling device pointer

.weight: The 'influence' of a particular cooling device on this zone. This is relative to the rest of the cooling devices. For example, if all cooling devices have a weight of 1, then they all contribute the same. You can use percentages if you want, but it's not mandatory. A weight of 0 means that this cooling device doesn't contribute to the cooling of this zone unless all cooling devices have a weight of 0. If all weights are 0, then they all contribute the same.

.trip_mask: This is a bit mask that gives the binding relation between this thermal zone and cdev, for a particular trip point. If nth bit is set, then the cdev and thermal zone are bound for trip point n.

.binding_limits: This is an array of cooling state limits. Must have exactly 2 * thermal_zone.number_of_trip_points. It is an array consisting of tuples <lower-state upper-state> of state limits. Each trip will be associated with one state limit tuple when binding. A NULL pointer means <THERMAL_NO_LIMITS THERMAL_NO_LIMITS> on all trips. These limits are used when binding a cdev to a trip point.

.match: This call back returns success(0) if the 'tz and cdev' need to be bound, as per platform data.

```
struct thermal_zone_params
```

This structure defines the platform level parameters for a thermal zone. This data, for each thermal zone should come from the platform layer. This is an optional feature where some platforms can choose not to provide this data.

.governor_name: Name of the thermal governor used for this zone

.no_hwmon: a boolean to indicate if the thermal to hwmon sysfs interface is required. when no_hwmon == false, a hwmon sysfs interface will be created. when no_hwmon == true, nothing will be done. In case the thermal_zone_params is NULL, the hwmon interface will be created (for backward compatibility).

.num_tbps: Number of thermal_bind_params entries for this zone

.tbp: thermal_bind_params entries

59.3.3 2. sysfs attributes structure

RO	read only value
WO	write only value
RW	read/write value

Thermal sysfs attributes will be represented under /sys/class/thermal. Hwmon sysfs I/F extension is also available under /sys/class/hwmon if hwmon is compiled in or built as a module.

Thermal zone device sys I/F, created once it' s registered:

```
/sys/class/thermal/thermal_zone[0-*]:
  |---type:                    Type of the thermal zone
  |---temp:                    Current temperature
  |---mode:                    Working mode of the thermal zone
  |---policy:                  Thermal governor used for this zone
  |---available_policies:      Available thermal governors for this zone
  |---trip_point_[0-*]_temp:   Trip point temperature
  |---trip_point_[0-*]_type:   Trip point type
  |---trip_point_[0-*]_hyst:   Hysteresis value for this trip point
  |---emul_temp:               Emulated temperature set node
  |---sustainable_power:       Sustainable dissipatable power
  |---k_po:                    Proportional term during temperature␣
↪overshoot
  |---k_pu:                    Proportional term during temperature␣
↪undershoot
  |---k_i:                     PID's integral term in the power allocator␣
↪gov
  |---k_d:                     PID's derivative term in the power allocator
  |---integral_cutoff:         Offset above which errors are accumulated
```

(continues on next page)

(continued from previous page)

```
  |---slope:                  Slope constant applied as linear␣
↪extrapolation
  |---offset:                 Offset constant applied as linear␣
↪extrapolation
```

Thermal cooling device sys I/F, created once it's registered:

```
/sys/class/thermal/cooling_device[0-*]:
  |---type:                   Type of the cooling device(processor/fan/...)
  |---max_state:              Maximum cooling state of the cooling device
  |---cur_state:              Current cooling state of the cooling device
  |---stats:                  Directory containing cooling device's␣
↪statistics
  |---stats/reset:            Writing any value resets the statistics
  |---stats/time_in_state_ms: Time (msec) spent in various cooling states
  |---stats/total_trans:      Total number of times cooling state is␣
↪changed
  |---stats/trans_table:      Cooing state transition table
```

Then next two dynamic attributes are created/removed in pairs. They represent the relationship between a thermal zone and its associated cooling device. They are created/removed for each successful execution of thermal_zone_bind_cooling_device/thermal_zone_unbind_cooling_device.

```
/sys/class/thermal/thermal_zone[0-*]:
  |---cdev[0-*]:              [0-*]th cooling device in current thermal␣
↪zone
  |---cdev[0-*]_trip_point:   Trip point that cdev[0-*] is associated with
  |---cdev[0-*]_weight:       Influence of the cooling device in
                              this thermal zone
```

Besides the thermal zone device sysfs I/F and cooling device sysfs I/F, the generic thermal driver also creates a hwmon sysfs I/F for each _type_ of thermal zone device. E.g. the generic thermal driver registers one hwmon class device and build the associated hwmon sysfs I/F for all the registered ACPI thermal zones.

```
/sys/class/hwmon/hwmon[0-*]:
  |---name:                   The type of the thermal zone devices
  |---temp[1-*]_input:        The current temperature of thermal zone [1-*]
  |---temp[1-*]_critical:     The critical trip point of thermal zone [1-*]
```

Please read Documentation/hwmon/sysfs-interface.rst for additional information.

Thermal zone attributes

type Strings which represent the thermal zone type. This is given by thermal zone driver as part of registration. E.g: "acpitz" indicates it's an ACPI thermal device. In order to keep it consistent with hwmon sys attribute; this should be a short, lowercase string, not containing spaces nor dashes. RO, Required

temp Current temperature as reported by thermal zone (sensor). Unit: millidegree Celsius RO, Required

mode One of the predefined values in [enabled, disabled]. This file gives information about the algorithm that is currently managing the thermal zone. It can be either default kernel based algorithm or user space application.

enabled enable Kernel Thermal management.

disabled Preventing kernel thermal zone driver actions upon trip points so that user application can take full charge of the thermal management.

RW, Optional

policy One of the various thermal governors used for a particular zone.

RW, Required

available_policies Available thermal governors which can be used for a particular zone.

RO, Required

trip_point_[0-*]_temp The temperature above which trip point will be fired.

Unit: millidegree Celsius

RO, Optional

trip_point_[0-*]_type Strings which indicate the type of the trip point.

E.g. it can be one of critical, hot, passive, active[0-*] for ACPI thermal zone.

RO, Optional

trip_point_[0-*]_hyst The hysteresis value for a trip point, represented as an integer Unit: Celsius RW, Optional

cdev[0-*] Sysfs link to the thermal cooling device node where the sys I/F for cooling device throttling control represents.

RO, Optional

cdev[0-*]_trip_point The trip point in this thermal zone which cdev[0-*] is associated with; -1 means the cooling device is not associated with any trip point.

RO, Optional

cdev[0-*]_weight The influence of cdev[0-*] in this thermal zone. This value is relative to the rest of cooling devices in the thermal zone. For example, if a cooling device has a weight double than that of other, it's twice as effective in cooling the thermal zone.

RW, Optional

passive Attribute is only present for zones in which the passive cooling policy is not supported by native thermal driver. Default is zero and can be set to a temperature (in millidegrees) to enable a passive trip point for the zone. Activation is done by polling with an interval of 1 second.

Unit: millidegrees Celsius

Valid values: 0 (disabled) or greater than 1000

RW, Optional

emul_temp Interface to set the emulated temperature method in thermal zone (sensor). After setting this temperature, the thermal zone may pass this temperature to platform emulation function if registered or cache it locally. This is useful in debugging different temperature threshold and its associated cooling action. This is write only node and writing 0 on this node should disable emulation. Unit: millidegree Celsius

WO, Optional

WARNING: Be careful while enabling this option on production systems, because userland can easily disable the thermal policy by simply flooding this sysfs node with low temperature values.

sustainable_power An estimate of the sustained power that can be dissipated by the thermal zone. Used by the power allocator governor. For more information see Documentation/driver-api/thermal/power_allocator.rst

Unit: milliwatts

RW, Optional

k_po The proportional term of the power allocator governor's PID controller during temperature overshoot. Temperature overshoot is when the current temperature is above the "desired temperature" trip point. For more information see Documentation/driver-api/thermal/power_allocator.rst

RW, Optional

k_pu The proportional term of the power allocator governor's PID controller during temperature undershoot. Temperature undershoot is when the current temperature is below the "desired temperature" trip point. For more information see Documentation/driver-api/thermal/power_allocator.rst

RW, Optional

k_i The integral term of the power allocator governor's PID controller. This term allows the PID controller to compensate for long term drift. For more information see Documentation/driver-api/thermal/power_allocator.rst

RW, Optional

k_d The derivative term of the power allocator governor's PID controller. For more information see Documentation/driver-api/thermal/power_allocator.rst

RW, Optional

integral_cutoff Temperature offset from the desired temperature trip point above which the integral term of the power allocator governor's PID controller starts accumulating errors. For example, if integral_cutoff is 0, then the integral term only accumulates error when temperature is above the desired temperature trip point. For more information see Documentation/driver-api/thermal/power_allocator.rst

Unit: millidegree Celsius

RW, Optional

slope The slope constant used in a linear extrapolation model to determine a hotspot temperature based off the sensor's raw readings. It is up to the device driver to determine the usage of these values.

RW, Optional

offset The offset constant used in a linear extrapolation model to determine a hotspot temperature based off the sensor's raw readings. It is up to the device driver to determine the usage of these values.

RW, Optional

Cooling device attributes

type String which represents the type of device, e.g:

- for generic ACPI: should be "Fan", "Processor" or "LCD"
- for memory controller device on intel_menlow platform: should be "Memory controller".

RO, Required

max_state The maximum permissible cooling state of this cooling device.

RO, Required

cur_state The current cooling state of this cooling device. The value can any integer numbers between 0 and max_state:

- cur_state == 0 means no cooling
- cur_state == max_state means the maximum cooling.

RW, Required

stats/reset Writing any value resets the cooling device's statistics. WO, Required

stats/time_in_state_ms: The amount of time spent by the cooling device in various cooling states. The output will have "<state> <time>" pair in each line, which will mean this cooling device spent <time> msec of time at <state>. Output will have one line for each of the supported states. usertime units here is 10mS (similar to other time exported in /proc). RO, Required

stats/total_trans: A single positive value showing the total number of times the state of a cooling device is changed.

RO, Required

stats/trans_table: This gives fine grained information about all the cooling state transitions. The cat output here is a two dimensional matrix, where an entry <i,j> (row i, column j) represents the number of transitions from State_i to State_j. If the transition table is bigger than PAGE_SIZE, reading this will return an -EFBIG error. RO, Required

59.3.4 3. A simple implementation

ACPI thermal zone may support multiple trip points like critical, hot, passive, active. If an ACPI thermal zone supports critical, passive, active[0] and active[1] at the same time, it may register itself as a thermal_zone_device (thermal_zone1) with 4 trip points in all. It has one processor and one fan, which are both registered as thermal_cooling_device. Both are considered to have the same effectiveness in cooling the thermal zone.

If the processor is listed in _PSL method, and the fan is listed in _AL0 method, the sys I/F structure will be built like this:

```
/sys/class/thermal:
 |thermal_zone1:
   |---type:                   acpitz
   |---temp:                   37000
   |---mode:                   enabled
   |---policy:                 step_wise
   |---available_policies:     step_wise fair_share
   |---trip_point_0_temp:      100000
   |---trip_point_0_type:      critical
   |---trip_point_1_temp:      80000
   |---trip_point_1_type:      passive
   |---trip_point_2_temp:      70000
   |---trip_point_2_type:      active0
   |---trip_point_3_temp:      60000
   |---trip_point_3_type:      active1
   |---cdev0:                  --->/sys/class/thermal/cooling_device0
   |---cdev0_trip_point:       1       /* cdev0 can be used for passive */
   |---cdev0_weight:           1024
   |---cdev1:                  --->/sys/class/thermal/cooling_device3
   |---cdev1_trip_point:       2       /* cdev1 can be used for active[0]*/
   |---cdev1_weight:           1024

 |cooling_device0:
   |---type:                   Processor
   |---max_state:              8
   |---cur_state:              0

 |cooling_device3:
   |---type:                   Fan
   |---max_state:              2
   |---cur_state:              0

/sys/class/hwmon:
 |hwmon0:
   |---name:                   acpitz
   |---temp1_input:            37000
   |---temp1_crit:             100000
```

59.3.5 4. Export Symbol APIs

4.1. get_tz_trend

This function returns the trend of a thermal zone, i.e the rate of change of temperature of the thermal zone. Ideally, the thermal sensor drivers are supposed to implement the callback. If they don' t, the thermal framework calculated the trend by comparing the previous and the current temperature values.

4.2. get_thermal_instance

This function returns the thermal_instance corresponding to a given {thermal_zone, cooling_device, trip_point} combination. Returns NULL if such an instance does not exist.

4.3. thermal_notify_framework

This function handles the trip events from sensor drivers. It starts throttling the cooling devices according to the policy configured. For CRITICAL and HOT trip points, this notifies the respective drivers, and does actual throttling for other trip points i.e ACTIVE and PASSIVE. The throttling policy is based on the configured platform data; if no platform data is provided, this uses the step_wise throttling policy.

4.4. thermal_cdev_update

This function serves as an arbitrator to set the state of a cooling device. It sets the cooling device to the deepest cooling state if possible.

59.3.6 5. thermal_emergency_poweroff

On an event of critical trip temperature crossing. Thermal framework allows the system to shutdown gracefully by calling orderly_poweroff(). In the event of a failure of orderly_poweroff() to shut down the system we are in danger of keeping the system alive at undesirably high temperatures. To mitigate this high risk scenario we program a work queue to fire after a pre-determined number of seconds to start an emergency shutdown of the device using the kernel_power_off() function. In case kernel_power_off() fails then finally emergency_restart() is called in the worst case.

The delay should be carefully profiled so as to give adequate time for orderly_poweroff(). In case of failure of an orderly_poweroff() the emergency poweroff kicks in after the delay has elapsed and shuts down the system.

If set to 0 emergency poweroff will not be supported. So a carefully profiled nonzero positive value is a must for emergerncy poweroff to be triggered.

59.4 Power allocator governor tunables

59.4.1 Trip points

The governor works optimally with the following two passive trip points:

1. “switch on” trip point: temperature above which the governor control loop starts operating. This is the first passive trip point of the thermal zone.
2. “desired temperature” trip point: it should be higher than the “switch on” trip point. This the target temperature the governor is controlling for. This is the last passive trip point of the thermal zone.

59.4.2 PID Controller

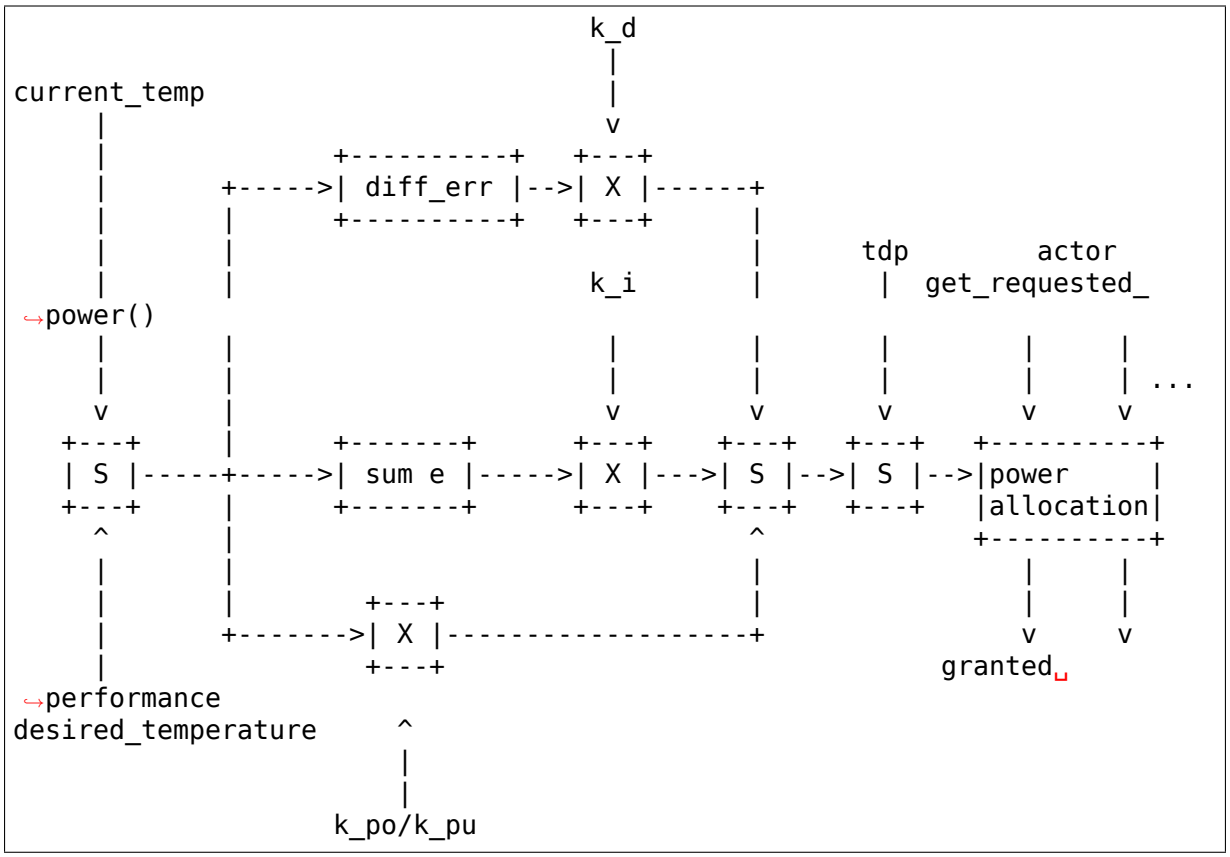
The power allocator governor implements a Proportional-Integral-Derivative controller (PID controller) with temperature as the control input and power as the controlled output:

P_max = k_p * e + k_i * err_integral + k_d * diff_err + sustainable_power

where

- e = desired_temperature - current_temperature
- err_integral is the sum of previous errors
- diff_err = e - previous_error

It is similar to the one depicted below:



59.4.3 Sustainable power

An estimate of the sustainable dissipatable power (in mW) should be provided while registering the thermal zone. This estimates the sustained power that can be dissipated at the desired control temperature. This is the maximum sustained power for allocation at the desired maximum temperature. The actual sustained power can vary for a number of reasons. The closed loop controller will take care of variations such as environmental conditions, and some factors related to the speed-grade of the silicon. sustainable_power is therefore simply an estimate, and may be tuned to affect the aggressiveness of the thermal ramp. For reference, the sustainable power of a 4”phone is typically 2000mW, while on a 10”tablet is around 4500mW (may vary depending on screen size).

If you are using device tree, do add it as a property of the thermal-zone. For example:

```
thermal-zones {
        soc_thermal {
                polling-delay = <1000>;
                polling-delay-passive = <100>;
                sustainable-power = <2500>;
                ...
```

Instead, if the thermal zone is registered from the platform code, pass a thermal_zone_params that has a sustainable_power. If no thermal_zone_params were being passed, then something like below will suffice:

```
static const struct thermal_zone_params tz_params = {
        .sustainable_power = 3500,
};
```

and then pass tz_params as the 5th parameter to thermal_zone_device_register()

59.4.4 k_po and k_pu

The implementation of the PID controller in the power allocator thermal governor allows the configuration of two proportional term constants: k_po and k_pu. k_po is the proportional term constant during temperature overshoot periods (current temperature is above “desired temperature” trip point). Conversely, k_pu is the proportional term constant during temperature undershoot periods (current temperature below “desired temperature” trip point).

These controls are intended as the primary mechanism for configuring the permitted thermal “ramp” of the system. For instance, a lower k_pu value will provide a slower ramp, at the cost of capping available capacity at a low temperature. On the other hand, a high value of k_pu will result in the governor granting very high power while temperature is low, and may lead to temperature overshooting.

The default value for k_pu is:

```
2 * sustainable_power / (desired_temperature - switch_on_temp)
```

This means that at switch_on_temp the output of the controller’s proportional term will be 2 * sustainable_power. The default value for k_po is:

```
sustainable_power / (desired_temperature - switch_on_temp)
```

Focusing on the proportional and feed forward values of the PID controller equation we have:

```
P_max = k_p * e + sustainable_power
```

The proportional term is proportional to the difference between the desired temperature and the current one. When the current temperature is the desired one, then the proportional component is zero and P_max = sustainable_power. That is, the system should operate in thermal equilibrium under constant load. sustainable_power is only an estimate, which is the reason for closed-loop control such as this.

Expanding k_pu we get:

```
P_max = 2 * sustainable_power * (T_set - T) / (T_set - T_on) +
    sustainable_power
```

where:

- T_set is the desired temperature
- T is the current temperature
- T_on is the switch on temperature

When the current temperature is the switch_on temperature, the above formula becomes:

```
P_max = 2 * sustainable_power * (T_set - T_on) / (T_set - T_on) +
    sustainable_power = 2 * sustainable_power + sustainable_power =
    3 * sustainable_power
```

Therefore, the proportional term alone linearly decreases power from 3 * sustainable_power to sustainable_power as the temperature rises from the switch on temperature to the desired temperature.

59.4.5 k_i and integral_cutoff

k_i configures the PID loop's integral term constant. This term allows the PID controller to compensate for long term drift and for the quantized nature of the output control: cooling devices can't set the exact power that the governor requests. When the temperature error is below integral_cutoff, errors are accumulated in the integral term. This term is then multiplied by k_i and the result added to the output of the controller. Typically k_i is set low (1 or 2) and integral_cutoff is 0.

59.4.6 k_d

k_d configures the PID loop's derivative term constant. It's recommended to leave it as the default: 0.

Cooling device power API

Cooling devices controlled by this governor must supply the additional "power" API in their cooling_device_ops. It consists on three ops:

1.
```
int get_requested_power(struct thermal_cooling_device *cdev,
                        struct thermal_zone_device *tz, u32 *power);
```

@cdev: The struct thermal_cooling_device pointer

@tz: thermal zone in which we are currently operating

@power: pointer in which to store the calculated power

get_requested_power() calculates the power requested by the device in milliwatts and stores it in @power . It should return 0 on success, -E* on failure. This is currently used by the power allocator governor to calculate how much power to give to each cooling device.

2.
```
int state2power(struct thermal_cooling_device *cdev, struct
                thermal_zone_device *tz, unsigned long state,
                u32 *power);
```

@cdev: The struct thermal_cooling_device pointer

@tz: thermal zone in which we are currently operating

@state: A cooling device state

@power: pointer in which to store the equivalent power

Convert cooling device state @state into power consumption in milliwatts and store it in @power. It should return 0 on success, -E* on failure. This is currently used by thermal core to calculate the maximum power that an actor can consume.

3.
```
int power2state(struct thermal_cooling_device *cdev, u32 power,
                unsigned long *state);
```

@cdev: The struct thermal_cooling_device pointer

@power: power in milliwatts

@state: pointer in which to store the resulting state

Calculate a cooling device state that would make the device consume at most @power mW and store it in @state. It should return 0 on success, -E* on failure. This is currently used by the thermal core to convert a given power set by the power allocator governor to a state that the cooling device can set. It is a function because this conversion may depend on external factors that may change so this function should the best conversion given "current circumstances".

59.4.7 Cooling device weights

Weights are a mechanism to bias the allocation among cooling devices. They express the relative power efficiency of different cooling devices. Higher weight can be used to express higher power efficiency. Weighting is relative such that if each cooling device has a weight of one they are considered equal. This is particularly useful in heterogeneous systems where two cooling devices may perform the same kind of compute, but with different efficiency. For example, a system with two different types of processors.

If the thermal zone is registered using thermal_zone_device_register() (i.e., platform code), then weights are passed as part of the thermal zone' s thermal_bind_parameters. If the platform is registered using device tree, then they are passed as the contribution property of each map in the cooling-maps node.

Limitations of the power allocator governor

The power allocator governor's PID controller works best if there is a periodic tick. If you have a driver that calls thermal_zone_device_update() (or anything that ends up calling the governor' s throttle() function) repetitively, the governor response won' t be very good. Note that this is not particular to this governor, step-wise will also misbehave if you call its throttle() faster than the normal thermal framework tick (due to interrupts for example) as it will overreact.

59.5 Kernel driver exynos_tmu

Supported chips:

- ARM Samsung Exynos4, Exynos5 series of SoC

  Datasheet: Not publicly available

Authors: Donggeun Kim <dg77.kim@samsung.com> Authors: Amit Daniel <amit.daniel@samsung.com>

59.5.1 TMU controller Description:

This driver allows to read temperature inside Samsung Exynos4/5 series of SoC.

The chip only exposes the measured 8-bit temperature code value through a register. Temperature can be taken from the temperature code. There are three equations converting from temperature to temperature code.

The three equations are:

1. Two point trimming:

```
Tc = (T - 25) * (TI2 - TI1) / (85 - 25) + TI1
```

2. One point trimming:

```
Tc = T + TI1 - 25
```

3. No trimming:

```
Tc = T + 50
```

Tc: Temperature code, T: Temperature,

TI1: Trimming info for 25 degree Celsius (stored at TRIMINFO register) Temperature code measured at 25 degree Celsius which is unchanged

TI2: Trimming info for 85 degree Celsius (stored at TRIMINFO register) Temperature code measured at 85 degree Celsius which is unchanged

TMU(Thermal Management Unit) in Exynos4/5 generates interrupt when temperature exceeds pre-defined levels. The maximum number of configurable threshold is five. The threshold levels are defined as follows:

```
Level_0: current temperature > trigger_level_0 + threshold
Level_1: current temperature > trigger_level_1 + threshold
Level_2: current temperature > trigger_level_2 + threshold
Level_3: current temperature > trigger_level_3 + threshold
```

The threshold and each trigger_level are set through the corresponding registers.

When an interrupt occurs, this driver notify kernel thermal framework with the function exynos_report_trigger. Although an interrupt condition for level_0 can be set, it can be used to synchronize the cooling action.

59.5.2 TMU driver description:

The exynos thermal driver is structured as:

```
                                  Kernel Core thermal framework
                           (thermal_core.c, step_wise.c, cpufreq_
↪cooling.c)
                                                          ^
                                                          |
                                                          |
TMU configuration data -----> TMU Driver  <----> Exynos Core thermal␣
↪wrapper
(exynos_tmu_data.c)          (exynos_tmu.c)       (exynos_thermal_common.c)
(exynos_tmu_data.h)          (exynos_tmu.h)       (exynos_thermal_common.h)
```

a) **TMU configuration data:** This consist of TMU register offsets/bitfields described through structure exynos_tmu_registers. Also several other platform data (struct exynos_tmu_platform_data) members are used to configure the TMU.

b) **TMU driver:** This component initialises the TMU controller and sets different thresholds. It invokes core thermal implementation with the call exynos_report_trigger.

c) **Exynos Core thermal wrapper:** This provides 3 wrapper function to use the Kernel core thermal framework. They

are exynos_unregister_thermal, exynos_register_thermal and exynos_report_trigger.

59.6 Exynos Emulation Mode

Copyright (C) 2012 Samsung Electronics

Written by Jonghwa Lee <jonghwa3.lee@samsung.com>

59.6.1 Description

Exynos 4x12 (4212, 4412) and 5 series provide emulation mode for thermal management unit. Thermal emulation mode supports software debug for TMU's operation. User can set temperature manually with software code and TMU will read current temperature from user value not from sensor's value.

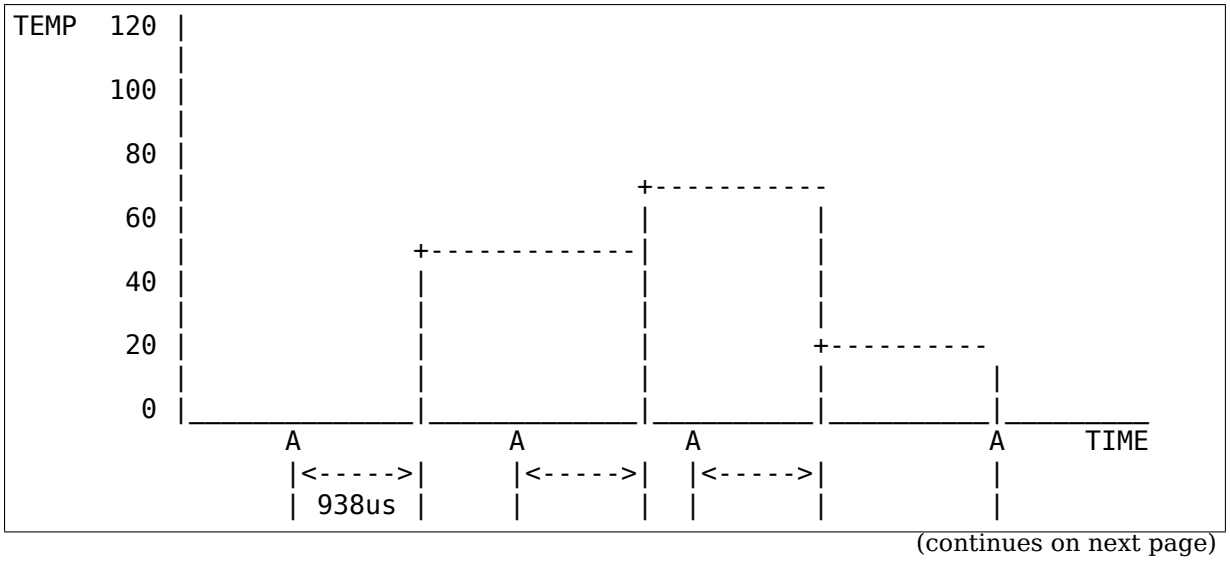
Enabling CONFIG_THERMAL_EMULATION option will make this support available. When it's enabled, sysfs node will be created as /sys/devices/virtual/thermal/thermal_zone'zone id'/emul_temp.

The sysfs node, 'emul_node', will contain value 0 for the initial state. When you input any temperature you want to update to sysfs node, it automatically enable emulation mode and current temperature will be changed into it.

(Exynos also supports user changeable delay time which would be used to delay of changing temperature. However, this node only uses same delay of real sensing time, 938us.)

Exynos emulation mode requires synchronous of value changing and enabling. It means when you want to update the any value of delay or next temperature, then you have to enable emulation mode at the same time. (Or you have to keep the mode enabling.) If you don't, it fails to change the value to updated one and just use last succeessful value repeatedly. That's why this node gives users the right to change termerpature only. Just one interface makes it more simply to use.

Disabling emulation mode only requires writing value 0 to sysfs node.



(continues on next page)

(continued from previous page)

```
emulation   : 0  50      |     70      |  20      |          0
current temp:   sensor   50            70         20        sensor
```

59.7 Intel Powerclamp Driver

By:

- Arjan van de Ven <arjan@linux.intel.com>
- Jacob Pan <jacob.jun.pan@linux.intel.com>

59.7.1 INTRODUCTION

Consider the situation where a system’s power consumption must be reduced at runtime, due to power budget, thermal constraint, or noise level, and where active cooling is not preferred. Software managed passive power reduction must be performed to prevent the hardware actions that are designed for catastrophic scenarios.

Currently, P-states, T-states (clock modulation), and CPU offlining are used for CPU throttling.

On Intel CPUs, C-states provide effective power reduction, but so far they’re only used opportunistically, based on workload. With the development of intel_powerclamp driver, the method of synchronizing idle injection across all online CPU threads was introduced. The goal is to achieve forced and controllable C-state residency.

Test/Analysis has been made in the areas of power, performance, scalability, and user experience. In many cases, clear advantage is shown over taking the CPU offline or modulating the CPU clock.

59.7.2 THEORY OF OPERATION

Idle Injection

On modern Intel processors (Nehalem or later), package level C-state residency is available in MSRs, thus also available to the kernel.

These MSRs are:

```
#define MSR_PKG_C2_RESIDENCY      0x60D
#define MSR_PKG_C3_RESIDENCY      0x3F8
#define MSR_PKG_C6_RESIDENCY      0x3F9
#define MSR_PKG_C7_RESIDENCY      0x3FA
```

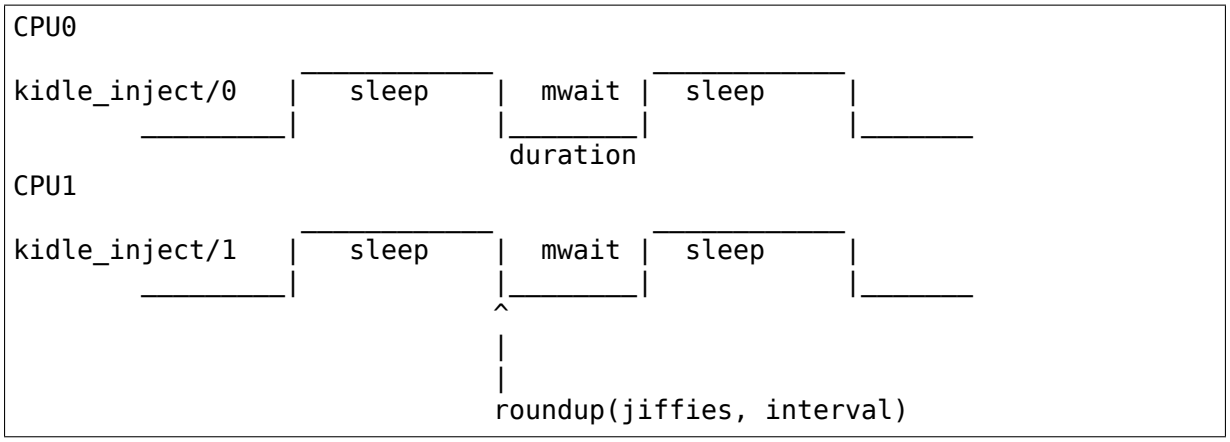
If the kernel can also inject idle time to the system, then a closed-loop control system can be established that manages package level C-state. The intel_powerclamp driver is conceived as such a control system, where the target set point is a user-selected idle ratio (based on power reduction), and the error is the difference between the actual package level C-state residency ratio and the target idle ratio.

Injection is controlled by high priority kernel threads, spawned for each online CPU.

These kernel threads, with SCHED_FIFO class, are created to perform clamping actions of controlled duty ratio and duration. Each per-CPU thread synchronizes its idle time and duration, based on the rounding of jiffies, so accumulated errors can be prevented to avoid a jittery effect. Threads are also bound to the CPU such that they cannot be migrated, unless the CPU is taken offline. In this case, threads belong to the offlined CPUs will be terminated immediately.

Running as SCHED_FIFO and relatively high priority, also allows such scheme to work for both preemptable and non-preemptable kernels. Alignment of idle time around jiffies ensures scalability for HZ values. This effect can be better visualized using a Perf timechart. The following diagram shows the behavior of kernel thread kidle_inject/cpu. During idle injection, it runs monitor/mwait idle for a given “duration”, then relinquishes the CPU to other tasks, until the next time interval.

The NOHZ schedule tick is disabled during idle time, but interrupts are not masked. Tests show that the extra wakeups from scheduler tick have a dramatic impact on the effectiveness of the powerclamp driver on large scale systems (Westmere system with 80 processors).



Only one CPU is allowed to collect statistics and update global control parameters. This CPU is referred to as the controlling CPU in this document. The controlling CPU is elected at runtime, with a policy that favors BSP, taking into account the possibility of a CPU hot-plug.

In terms of dynamics of the idle control system, package level idle time is considered largely as a non-causal system where its behavior cannot be based on the past or current input. Therefore, the intel_powerclamp driver attempts to enforce the desired idle time instantly as given input (target idle ratio). After injection, powerclamp monitors the actual idle for a given time window and adjust the next injection accordingly to avoid over/under correction.

When used in a causal control system, such as a temperature control, it is up to the user of this driver to implement algorithms where past samples and outputs are included in the feedback. For example, a PID-based thermal controller can use the powerclamp driver to maintain a desired target temperature, based on integral and derivative gains of the past samples.

Calibration

During scalability testing, it is observed that synchronized actions among CPUs become challenging as the number of cores grows. This is also true for the ability of a system to enter package level C-states.

To make sure the intel_powerclamp driver scales well, online calibration is implemented. The goals for doing such a calibration are:

a) determine the effective range of idle injection ratio

b) determine the amount of compensation needed at each target ratio

Compensation to each target ratio consists of two parts:

> a) steady state error compensation This is to offset the error occurring when the system can enter idle without extra wakeups (such as external interrupts).
>
> b) dynamic error compensation When an excessive amount of wakeups occurs during idle, an additional idle ratio can be added to quiet interrupts, by slowing down CPU activities.

A debugfs file is provided for the user to examine compensation progress and results, such as on a Westmere system:

```
[jacob@nex01 ~]$ cat
/sys/kernel/debug/intel_powerclamp/powerclamp_calib
controlling cpu: 0
pct confidence steady dynamic (compensation)
0       0       0       0
1       1       0       0
2       1       1       0
3       3       1       0
4       3       1       0
5       3       1       0
6       3       1       0
7       3       1       0
8       3       1       0
...
30      3       2       0
31      3       2       0
32      3       1       0
33      3       2       0
34      3       1       0
35      3       2       0
36      3       1       0
37      3       2       0
38      3       1       0
39      3       2       0
40      3       3       0
41      3       1       0
42      3       2       0
43      3       1       0
44      3       1       0
45      3       2       0
46      3       3       0
47      3       0       0
```

(continues on next page)

(continued from previous page)

```
48      3       2       0
49      3       3       0
```

Calibration occurs during runtime. No offline method is available. Steady state compensation is used only when confidence levels of all adjacent ratios have reached satisfactory level. A confidence level is accumulated based on clean data collected at runtime. Data collected during a period without extra interrupts is considered clean.

To compensate for excessive amounts of wakeup during idle, additional idle time is injected when such a condition is detected. Currently, we have a simple algorithm to double the injection ratio. A possible enhancement might be to throttle the offending IRQ, such as delaying EOI for level triggered interrupts. But it is a challenge to be non-intrusive to the scheduler or the IRQ core code.

CPU Online/Offline

Per-CPU kernel threads are started/stopped upon receiving notifications of CPU hotplug activities. The intel_powerclamp driver keeps track of clamping kernel threads, even after they are migrated to other CPUs, after a CPU offline event.

59.7.3 Performance Analysis

This section describes the general performance data collected on multiple systems, including Westmere (80P) and Ivy Bridge (4P, 8P).

Effectiveness and Limitations

The maximum range that idle injection is allowed is capped at 50 percent. As mentioned earlier, since interrupts are allowed during forced idle time, excessive interrupts could result in less effectiveness. The extreme case would be doing a ping -f to generated flooded network interrupts without much CPU acknowledgement. In this case, little can be done from the idle injection threads. In most normal cases, such as scp a large file, applications can be throttled by the powerclamp driver, since slowing down the CPU also slows down network protocol processing, which in turn reduces interrupts.

When control parameters change at runtime by the controlling CPU, it may take an additional period for the rest of the CPUs to catch up with the changes. During this time, idle injection is out of sync, thus not able to enter package C- states at the expected ratio. But this effect is minor, in that in most cases change to the target ratio is updated much less frequently than the idle injection frequency.

Scalability

Tests also show a minor, but measurable, difference between the 4P/8P Ivy Bridge system and the 80P Westmere server under 50% idle ratio. More compensation is needed on Westmere for the same amount of target idle ratio. The compensation also increases as the idle ratio gets larger. The above reason constitutes the need for the calibration code.

On the IVB 8P system, compared to an offline CPU, powerclamp can achieve up to 40% better performance per watt. (measured by a spin counter summed over per CPU counting threads spawned for all running CPUs).

59.7.4 Usage and Interfaces

The powerclamp driver is registered to the generic thermal layer as a cooling device. Currently, it's not bound to any thermal zones:

```
jacob@chromoly:/sys/class/thermal/cooling_device14$ grep . *
cur_state:0
max_state:50
type:intel_powerclamp
```

cur_state allows user to set the desired idle percentage. Writing 0 to cur_state will stop idle injection. Writing a value between 1 and max_state will start the idle injection. Reading cur_state returns the actual and current idle percentage. This may not be the same value set by the user in that current idle percentage depends on workload and includes natural idle. When idle injection is disabled, reading cur_state returns value -1 instead of 0 which is to avoid confusing 100% busy state with the disabled state.

Example usage: - To inject 25% idle time:

```
$ sudo sh -c "echo 25 > /sys/class/thermal/cooling_device80/cur_state
```

If the system is not busy and has more than 25% idle time already, then the powerclamp driver will not start idle injection. Using Top will not show idle injection kernel threads.

If the system is busy (spin test below) and has less than 25% natural idle time, powerclamp kernel threads will do idle injection. Forced idle time is accounted as normal idle in that common code path is taken as the idle task.

In this example, 24.1% idle is shown. This helps the system admin or user determine the cause of slowdown, when a powerclamp driver is in action:

```
Tasks: 197 total,   1 running, 196 sleeping,   0 stopped,   0 zombie
Cpu(s): 71.2%us,  4.7%sy,  0.0%ni, 24.1%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0
↪%st
Mem:   3943228k total,  1689632k used,  2253596k free,    74960k buffers
Swap:  4087804k total,        0k used,  4087804k free,   945336k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3352 jacob     20   0  262m  644  428 S  286  0.0   0:17.16 spin
 3341 root     -51   0     0    0    0 D   25  0.0   0:01.62 kidle_inject/0
```

(continues on next page)

(continued from previous page)

```
 3344 root     -51   0     0    0    0 D   25  0.0   0:01.60 kidle_inject/3
 3342 root     -51   0     0    0    0 D   25  0.0   0:01.61 kidle_inject/1
 3343 root     -51   0     0    0    0 D   25  0.0   0:01.60 kidle_inject/2
 2935 jacob     20   0  696m 125m  35m S    5  3.3   0:31.11 firefox
 1546 root      20   0  158m  20m 6640 S    3  0.5   0:26.97 Xorg
 2100 jacob     20   0 1223m  88m  30m S    3  2.3   0:23.68 compiz
```

Tests have shown that by using the powerclamp driver as a cooling device, a PID based userspace thermal controller can manage to control CPU temperature effectively, when no other thermal influence is added. For example, a UltraBook user can compile the kernel under certain temperature (below most active trip points).

59.8 Kernel driver nouveau

Supported chips:

- NV43+

Authors: Martin Peres (mupuf) <martin.peres@free.fr>

59.8.1 Description

This driver allows to read the GPU core temperature, drive the GPU fan and set temperature alarms.

Currently, due to the absence of in-kernel API to access HWMON drivers, Nouveau cannot access any of the i2c external monitoring chips it may find. If you have one of those, temperature and/or fan management through Nouveau's HWMON interface is likely not to work. This document may then not cover your situation entirely.

59.8.2 Temperature management

Temperature is exposed under as a read-only HWMON attribute temp1_input.

In order to protect the GPU from overheating, Nouveau supports 4 configurable temperature thresholds:

- **Fan_boost:** Fan speed is set to 100% when reaching this temperature;
- **Downclock:** The GPU will be downclocked to reduce its power dissipation;
- **Critical:** The GPU is put on hold to further lower power dissipation;
- **Shutdown:** Shut the computer down to protect your GPU.

WARNING: Some of these thresholds may not be used by Nouveau depending on your chipset.

The default value for these thresholds comes from the GPU's vbios. These thresholds can be configured thanks to the following HWMON attributes:

- Fan_boost: temp1_auto_point1_temp and temp1_auto_point1_temp_hyst;

- Downclock: temp1_max and temp1_max_hyst;
- Critical: temp1_crit and temp1_crit_hyst;
- Shutdown: temp1_emergency and temp1_emergency_hyst.

NOTE: Remember that the values are stored as milli degrees Celsius. Don't forget to multiply!

59.8.3 Fan management

Not all cards have a drivable fan. If you do, then the following HWMON attributes should be available:

- **pwm1_enable:** Current fan management mode (NONE, MANUAL or AUTO);
- **pwm1:** Current PWM value (power percentage);
- **pwm1_min:** The minimum PWM speed allowed;
- **pwm1_max:** The maximum PWM speed allowed (bypassed when hitting Fan_boost);

You may also have the following attribute:

- **fan1_input:** Speed in RPM of your fan.

Your fan can be driven in different modes:

- 0: The fan is left untouched;
- 1: The fan can be driven in manual (use pwm1 to change the speed);
- 2; The fan is driven automatically depending on the temperature.

NOTE: Be sure to use the manual mode if you want to drive the fan speed manually

NOTE2: When operating in manual mode outside the vbios-defined [PWM_min, PWM_max] range, the reported fan speed (RPM) may not be accurate depending on your hardware.

59.8.4 Bug reports

Thermal management on Nouveau is new and may not work on all cards. If you have inquiries, please ping mupuf on IRC (#nouveau, freenode).

Bug reports should be filled on Freedesktop's bug tracker. Please follow http://nouveau.freedesktop.org/wiki/Bugs

59.9 Kernel driver: x86_pkg_temp_thermal

Supported chips:

- x86: with package level thermal management

(Verify using: CPUID.06H:EAX[bit 6] =1)

Authors: Srinivas Pandruvada <srinivas.pandruvada@linux.intel.com>

59.9.1 Reference

Intel® 64 and IA-32 Architectures Software Developer’ s Manual (Jan, 2013): Chapter 14.6: PACKAGE LEVEL THERMAL MANAGEMENT

59.9.2 Description

This driver register CPU digital temperature package level sensor as a thermal zone with maximum two user mode configurable trip points. Number of trip points depends on the capability of the package. Once the trip point is violated, user mode can receive notification via thermal notification mechanism and can take any action to control temperature.

59.9.3 Threshold management

Each package will register as a thermal zone under /sys/class/thermal.

Example:

```
/sys/class/thermal/thermal_zone1
```

This contains two trip points:

- trip_point_0_temp
- trip_point_1_temp

User can set any temperature between 0 to TJ-Max temperature. Temperature units are in milli-degree Celsius. Refer to “Documentation/driver-api/thermal/sysfs-api.rst” for thermal sys-fs details.

Any value other than 0 in these trip points, can trigger thermal notifications. Setting 0, stops sending thermal notifications.

Thermal notifications: To get kobject-uevent notifications, set the thermal zone policy to “user_space” .

For example:

```
echo -n "user_space" > policy
```

CHAPTER

SIXTY

FPGA SUBSYSTEM

Author Alan Tull

60.1 Introduction

The FPGA subsystem supports reprogramming FPGAs dynamically under Linux. Some of the core intentions of the FPGA subsystems are:

- The FPGA subsystem is vendor agnostic.
- The FPGA subsystem separates upper layers (userspace interfaces and enumeration) from lower layers that know how to program a specific FPGA.
- Code should not be shared between upper and lower layers. This should go without saying. If that seems necessary, there's probably framework functionality that can be added that will benefit other users. Write the linux-fpga mailing list and maintainers and seek out a solution that expands the framework for broad reuse.
- Generally, when adding code, think of the future. Plan for reuse.

The framework in the kernel is divided into:

60.1.1 FPGA Manager

If you are adding a new FPGA or a new method of programming an FPGA, this is the subsystem for you. Low level FPGA manager drivers contain the knowledge of how to program a specific device. This subsystem includes the framework in fpga-mgr.c and the low level drivers that are registered with it.

60.1.2 FPGA Bridge

FPGA Bridges prevent spurious signals from going out of an FPGA or a region of an FPGA during programming. They are disabled before programming begins and re-enabled afterwards. An FPGA bridge may be actual hard hardware that gates a bus to a CPU or a soft ("freeze") bridge in FPGA fabric that surrounds a partial reconfiguration region of an FPGA. This subsystem includes fpga-bridge.c and the low level drivers that are registered with it.

60.1.3 FPGA Region

If you are adding a new interface to the FPGA framework, add it on top of an FPGA region.

The FPGA Region framework (fpga-region.c) associates managers and bridges as reconfigurable regions. A region may refer to the whole FPGA in full reconfiguration or to a partial reconfiguration region.

The Device Tree FPGA Region support (of-fpga-region.c) handles reprogramming FPGAs when device tree overlays are applied.

60.2 FPGA Manager

60.2.1 Overview

The FPGA manager core exports a set of functions for programming an FPGA with an image. The API is manufacturer agnostic. All manufacturer specifics are hidden away in a low level driver which registers a set of ops with the core. The FPGA image data itself is very manufacturer specific, but for our purposes it's just binary data. The FPGA manager core won't parse it.

The FPGA image to be programmed can be in a scatter gather list, a single contiguous buffer, or a firmware file. Because allocating contiguous kernel memory for the buffer should be avoided, users are encouraged to use a scatter gather list instead if possible.

The particulars for programming the image are presented in a structure (struct fpga_image_info). This struct contains parameters such as pointers to the FPGA image as well as image-specific particulars such as whether the image was built for full or partial reconfiguration.

60.2.2 How to support a new FPGA device

To add another FPGA manager, write a driver that implements a set of ops. The probe function calls `fpga_mgr_register()`, such as:

```
static const struct fpga_manager_ops socfpga_fpga_ops = {
        .write_init = socfpga_fpga_ops_configure_init,
        .write = socfpga_fpga_ops_configure_write,
        .write_complete = socfpga_fpga_ops_configure_complete,
        .state = socfpga_fpga_ops_state,
};

static int socfpga_fpga_probe(struct platform_device *pdev)
{
        struct device *dev = &pdev->dev;
        struct socfpga_fpga_priv *priv;
        struct fpga_manager *mgr;
        int ret;

        priv = devm_kzalloc(dev, sizeof(*priv), GFP_KERNEL);
```

(continues on next page)

(continued from previous page)

```
        if (!priv)
                return -ENOMEM;

        /*
         * do ioremaps, get interrupts, etc. and save
         * them in priv
         */

        mgr = devm_fpga_mgr_create(dev, "Altera SOCFPGA FPGA Manager",
                                   &socfpga_fpga_ops, priv);
        if (!mgr)
                return -ENOMEM;

        platform_set_drvdata(pdev, mgr);

        return fpga_mgr_register(mgr);
}

static int socfpga_fpga_remove(struct platform_device *pdev)
{
        struct fpga_manager *mgr = platform_get_drvdata(pdev);

        fpga_mgr_unregister(mgr);

        return 0;
}
```

The ops will implement whatever device specific register writes are needed to do the programming sequence for this particular FPGA. These ops return 0 for success or negative error codes otherwise.

The programming sequence is::

1. .write_init
2. .write or .write_sg (may be called once or multiple times)
3. .write_complete

The .write_init function will prepare the FPGA to receive the image data. The buffer passed into .write_init will be at most .initial_header_size bytes long; if the whole bitstream is not immediately available then the core code will buffer up at least this much before starting.

The .write function writes a buffer to the FPGA. The buffer may be contain the whole FPGA image or may be a smaller chunk of an FPGA image. In the latter case, this function is called multiple times for successive chunks. This interface is suitable for drivers which use PIO.

The .write_sg version behaves the same as .write except the input is a sg_table scatter list. This interface is suitable for drivers which use DMA.

The .write_complete function is called after all the image has been written to put the FPGA into operating mode.

The ops include a .state function which will determine the state the FPGA is in and return a code of type enum fpga_mgr_states. It doesn't result in a change in state.

60.2.3 API for implementing a new FPGA Manager driver

- `fpga_mgr_states` —Values for `fpga_manager->state`.
- struct `fpga_manager` —the FPGA manager struct
- struct `fpga_manager_ops` —Low level FPGA manager driver ops
- `devm_fpga_mgr_create()` —Allocate and init a manager struct
- `fpga_mgr_register()` —Register an FPGA manager
- `fpga_mgr_unregister()` —Unregister an FPGA manager

enum **fpga_mgr_states**
fpga framework states

Constants

FPGA_MGR_STATE_UNKNOWN can't determine state

FPGA_MGR_STATE_POWER_OFF FPGA power is off

FPGA_MGR_STATE_POWER_UP FPGA reports power is up

FPGA_MGR_STATE_RESET FPGA in reset state

FPGA_MGR_STATE_FIRMWARE_REQ firmware request in progress

FPGA_MGR_STATE_FIRMWARE_REQ_ERR firmware request failed

FPGA_MGR_STATE_WRITE_INIT preparing FPGA for programming

FPGA_MGR_STATE_WRITE_INIT_ERR Error during WRITE_INIT stage

FPGA_MGR_STATE_WRITE writing image to FPGA

FPGA_MGR_STATE_WRITE_ERR Error while writing FPGA

FPGA_MGR_STATE_WRITE_COMPLETE Doing post programming steps

FPGA_MGR_STATE_WRITE_COMPLETE_ERR Error during WRITE_COMPLETE

FPGA_MGR_STATE_OPERATING FPGA is programmed and operating

struct **fpga_manager**
fpga manager structure

Definition

```
struct fpga_manager {
  const char *name;
  struct device dev;
  struct mutex ref_mutex;
  enum fpga_mgr_states state;
  struct fpga_compat_id *compat_id;
  const struct fpga_manager_ops *mops;
  void *priv;
};
```

Members

name name of low level fpga manager

dev fpga manager device

ref_mutex only allows one reference to fpga manager

state state of fpga manager

compat_id FPGA manager id for compatibility check.

mops pointer to struct of fpga manager ops

priv low level driver private date

struct **fpga_manager_ops**
ops for low level fpga manager drivers

Definition

```
struct fpga_manager_ops {
  size_t initial_header_size;
  enum fpga_mgr_states (*state)(struct fpga_manager *mgr);
  u64 (*status)(struct fpga_manager *mgr);
  int (*write_init)(struct fpga_manager *mgr,struct fpga_image_info *info,␣
↪const char *buf, size_t count);
  int (*write)(struct fpga_manager *mgr, const char *buf, size_t count);
  int (*write_sg)(struct fpga_manager *mgr, struct sg_table *sgt);
  int (*write_complete)(struct fpga_manager *mgr, struct fpga_image_info␣
↪*info);
  void (*fpga_remove)(struct fpga_manager *mgr);
  const struct attribute_group **groups;
};
```

Members

initial_header_size Maximum number of bytes that should be passed into write_init

state returns an enum value of the FPGA's state

status returns status of the FPGA, including reconfiguration error code

write_init prepare the FPGA to receive confuration data

write write count bytes of configuration data to the FPGA

write_sg write the scatter list of configuration data to the FPGA

write_complete set FPGA to operating state after writing is done

fpga_remove optional: Set FPGA into a specific state during driver remove

groups optional attribute groups.

Description

fpga_manager_ops are the low level functions implemented by a specific fpga manager driver. The optional ones are tested for NULL before being called, so leaving them out is fine.

struct fpga_manager * **devm_fpga_mgr_create**(struct device * dev, const char * name, const struct fpga_manager_ops * mops, void * priv)
create and initialize a managed FPGA manager struct

Parameters

struct device * dev fpga manager device from pdev

const char * name fpga manager name

const struct fpga_manager_ops * mops pointer to structure of fpga manager ops

void * priv fpga manager private data

Description

This function is intended for use in a FPGA manager driver's probe function. After the manager driver creates the manager struct with `devm_fpga_mgr_create()`, it should register it with `fpga_mgr_register()`. The manager driver's remove function should call `fpga_mgr_unregister()`. The manager struct allocated with this function will be freed automatically on driver detach. This includes the case of a probe function returning error before calling `fpga_mgr_register()`, the struct will still get cleaned up.

Return

pointer to struct fpga_manager or NULL

int **fpga_mgr_register**(struct fpga_manager * mgr)
register a FPGA manager

Parameters

struct fpga_manager * mgr fpga manager struct

Return

0 on success, negative error code otherwise.

void **fpga_mgr_unregister**(struct fpga_manager * mgr)
unregister a FPGA manager

Parameters

struct fpga_manager * mgr fpga manager struct

Description

This function is intended for use in a FPGA manager driver's remove function.

60.3 FPGA Bridge

60.3.1 API to implement a new FPGA bridge

- struct `fpga_bridge` —The FPGA Bridge structure
- struct `fpga_bridge_ops` —Low level Bridge driver ops
- `devm_fpga_bridge_create()` —Allocate and init a bridge struct
- `fpga_bridge_register()` —Register a bridge
- `fpga_bridge_unregister()` —Unregister a bridge

struct **fpga_bridge**
FPGA bridge structure

Definition

```
struct fpga_bridge {
  const char *name;
  struct device dev;
  struct mutex mutex;
  const struct fpga_bridge_ops *br_ops;
  struct fpga_image_info *info;
  struct list_head node;
  void *priv;
};
```

Members

name name of low level FPGA bridge

dev FPGA bridge device

mutex enforces exclusive reference to bridge

br_ops pointer to struct of FPGA bridge ops

info fpga image specific information

node FPGA bridge list node

priv low level driver private date

struct **fpga_bridge_ops**
ops for low level FPGA bridge drivers

Definition

```
struct fpga_bridge_ops {
  int (*enable_show)(struct fpga_bridge *bridge);
  int (*enable_set)(struct fpga_bridge *bridge, bool enable);
  void (*fpga_bridge_remove)(struct fpga_bridge *bridge);
  const struct attribute_group **groups;
};
```

Members

enable_show returns the FPGA bridge's status

enable_set set a FPGA bridge as enabled or disabled

fpga_bridge_remove set FPGA into a specific state during driver remove

groups optional attribute groups.

struct fpga_bridge * **devm_fpga_bridge_create**(struct device * dev, const char * name, const struct fpga_bridge_ops * br_ops, void * priv)
create and init a managed struct fpga_bridge

Parameters

struct device * dev FPGA bridge device from pdev

const char * name FPGA bridge name

const struct fpga_bridge_ops * br_ops pointer to structure of fpga bridge ops

void * priv FPGA bridge private data

Description

This function is intended for use in a FPGA bridge driver's probe function. After the bridge driver creates the struct with `devm_fpga_bridge_create()`, it should register the bridge with `fpga_bridge_register()`. The bridge driver's remove function should call `fpga_bridge_unregister()`. The bridge struct allocated with this function will be freed automatically on driver detach. This includes the case of a probe function returning error before calling `fpga_bridge_register()`, the struct will still get cleaned up.

Return

struct fpga_bridge or NULL

int **fpga_bridge_register**(struct fpga_bridge * bridge)
register a FPGA bridge

Parameters

struct fpga_bridge * bridge FPGA bridge struct

Return

0 for success, error code otherwise.

void **fpga_bridge_unregister**(struct fpga_bridge * bridge)
unregister a FPGA bridge

Parameters

struct fpga_bridge * bridge FPGA bridge struct

Description

This function is intended for use in a FPGA bridge driver's remove function.

60.4 FPGA Region

60.4.1 Overview

This document is meant to be a brief overview of the FPGA region API usage. A more conceptual look at regions can be found in the Device Tree binding document[1].

For the purposes of this API document, let's just say that a region associates an FPGA Manager and a bridge (or bridges) with a reprogrammable region of an FPGA or the whole FPGA. The API provides a way to register a region and to program a region.

[1] ../devicetree/bindings/fpga/fpga-region.txt

Currently the only layer above fpga-region.c in the kernel is the Device Tree support (of-fpga-region.c) described in[1]. The DT support layer uses regions to program the FPGA and then DT to handle enumeration. The common region code is intended to be used by other schemes that have other ways of accomplishing enumeration after programming.

An fpga-region can be set up to know the following things:

- which FPGA manager to use to do the programming
- which bridges to disable before programming and enable afterwards.

Additional info needed to program the FPGA image is passed in the struct fpga_image_info including:

- pointers to the image as either a scatter-gather buffer, a contiguous buffer, or the name of firmware file
- flags indicating specifics such as whether the image is for partial reconfiguration.

60.4.2 How to add a new FPGA region

An example of usage can be seen in the probe function of[2].

60.4.3 API to add a new FPGA region

- struct `fpga_region` —The FPGA region struct
- `devm_fpga_region_create()` —Allocate and init a region struct
- `fpga_region_register()` —Register an FPGA region
- `fpga_region_unregister()` —Unregister an FPGA region

The FPGA region's probe function will need to get a reference to the FPGA Manager it will be using to do the programming. This usually would happen during the region's probe function.

- `fpga_mgr_get()` —Get a reference to an FPGA manager, raise ref count
- `of_fpga_mgr_get()` —Get a reference to an FPGA manager, raise ref count, given a device node.
- `fpga_mgr_put()` —Put an FPGA manager

The FPGA region will need to specify which bridges to control while programming the FPGA. The region driver can build a list of bridges during probe time (`fpga_region->bridge_list`) or it can have a function that creates the list of bridges to program just before programming (`fpga_region->get_bridges`). The FPGA bridge framework supplies the following APIs to handle building or tearing down that list.

- `fpga_bridge_get_to_list()` —Get a ref of an FPGA bridge, add it to a list

[2] ../../drivers/fpga/of-fpga-region.c

- `of_fpga_bridge_get_to_list()` —Get a ref of an FPGA bridge, add it to a list, given a device node
- `fpga_bridges_put()` —Given a list of bridges, put them

struct **fpga_region**
FPGA Region structure

Definition

```
struct fpga_region {
  struct device dev;
  struct mutex mutex;
  struct list_head bridge_list;
  struct fpga_manager *mgr;
  struct fpga_image_info *info;
  struct fpga_compat_id *compat_id;
  void *priv;
  int (*get_bridges)(struct fpga_region *region);
};
```

Members

dev FPGA Region device

mutex enforces exclusive reference to region

bridge_list list of FPGA bridges specified in region

mgr FPGA manager

info FPGA image info

compat_id FPGA region id for compatibility check.

priv private data

get_bridges optional function to get bridges to a list

struct fpga_region * **devm_fpga_region_create**(struct device * dev, struct fpga_manager * mgr, int (*get_bridges)(struct fpga_region *))
create and initialize a managed FPGA region struct

Parameters

struct device * dev device parent

struct fpga_manager * mgr manager that programs this region

int (*)(struct fpga_region *) get_bridges optional function to get bridges to a list

Description

This function is intended for use in a FPGA region driver's probe function. After the region driver creates the region struct with `devm_fpga_region_create()`, it should register it with `fpga_region_register()`. The region driver's remove function should call `fpga_region_unregister()`. The region struct allocated with this function will be freed automatically on driver detach. This includes the case

of a probe function returning error before calling `fpga_region_register()`, the struct will still get cleaned up.

Return

struct fpga_region or NULL

int **fpga_region_register**(struct fpga_region * region)
register a FPGA region

Parameters

struct fpga_region * region FPGA region

Return

0 or -errno

void **fpga_region_unregister**(struct fpga_region * region)
unregister a FPGA region

Parameters

struct fpga_region * region FPGA region

Description

This function is intended for use in a FPGA region driver's remove function.

struct fpga_manager * **fpga_mgr_get**(struct device * dev)
Given a device, get a reference to a fpga mgr.

Parameters

struct device * dev parent device that fpga mgr was registered with

Return

fpga manager struct or IS_ERR() condition containing error code.

struct fpga_manager * **of_fpga_mgr_get**(struct device_node * node)
Given a device node, get a reference to a fpga mgr.

Parameters

struct device_node * node device node

Return

fpga manager struct or IS_ERR() condition containing error code.

void **fpga_mgr_put**(struct fpga_manager * mgr)
release a reference to a fpga manager

Parameters

struct fpga_manager * mgr fpga manager structure

int **fpga_bridge_get_to_list**(struct device * dev, struct fpga_image_info * info, struct list_head * bridge_list)
given device, get a bridge, add it to a list

Parameters

struct device * dev FPGA bridge device

struct fpga_image_info * info fpga image specific information

struct list_head * bridge_list list of FPGA bridges

Description

Get an exclusive reference to the bridge and and it to the list.

Return 0 for success, error code from fpga_bridge_get() othewise.

int **of_fpga_bridge_get_to_list**(struct device_node * np, struct fpga_image_info * info, struct list_head * bridge_list)
get a bridge, add it to a list

Parameters

struct device_node * np node pointer of a FPGA bridge

struct fpga_image_info * info fpga image specific information

struct list_head * bridge_list list of FPGA bridges

Description

Get an exclusive reference to the bridge and and it to the list.

Return 0 for success, error code from of_fpga_bridge_get() othewise.

void **fpga_bridges_put**(struct list_head * bridge_list)
put bridges

Parameters

struct list_head * bridge_list list of FPGA bridges

Description

For each bridge in the list, put the bridge and remove it from the list. If list is empty, do nothing.

60.5 In-kernel API for FPGA Programming

60.5.1 Overview

The in-kernel API for FPGA programming is a combination of APIs from FPGA manager, bridge, and regions. The actual function used to trigger FPGA programming is `fpga_region_program_fpga()`.

`fpga_region_program_fpga()` uses functionality supplied by the FPGA manager and bridges. It will:

- lock the region's mutex
- lock the mutex of the region's FPGA manager
- build a list of FPGA bridges if a method has been specified to do so
- disable the bridges
- program the FPGA using info passed in `fpga_region->info`.

- re-enable the bridges
- release the locks

The struct fpga_image_info specifies what FPGA image to program. It is allocated/freed by `fpga_image_info_alloc()` and freed with `fpga_image_info_free()`

60.5.2 How to program an FPGA using a region

When the FPGA region driver probed, it was given a pointer to an FPGA manager driver so it knows which manager to use. The region also either has a list of bridges to control during programming or it has a pointer to a function that will generate that list. Here's some sample code of what to do next:

```
#include <linux/fpga/fpga-mgr.h>
#include <linux/fpga/fpga-region.h>

struct fpga_image_info *info;
int ret;

/*
 * First, alloc the struct with information about the FPGA image to
 * program.
 */
info = fpga_image_info_alloc(dev);
if (!info)
        return -ENOMEM;

/* Set flags as needed, such as: */
info->flags = FPGA_MGR_PARTIAL_RECONFIG;

/*
 * Indicate where the FPGA image is. This is pseudo-code; you're
 * going to use one of these three.
 */
if (image is in a scatter gather table) {

        info->sgt = [your scatter gather table]

} else if (image is in a buffer) {

        info->buf = [your image buffer]
        info->count = [image buffer size]

} else if (image is in a firmware file) {

        info->firmware_name = devm_kstrdup(dev, firmware_name,
                                           GFP_KERNEL);

}

/* Add info to region and do the programming */
region->info = info;
ret = fpga_region_program_fpga(region);
```

(continues on next page)

(continued from previous page)

```
/* Deallocate the image info if you're done with it */
region->info = NULL;
fpga_image_info_free(info);

if (ret)
        return ret;

/* Now enumerate whatever hardware has appeared in the FPGA. */
```

60.5.3 API for programming an FPGA

- `fpga_region_program_fpga()` —Program an FPGA
- `fpga_image_info` —Specifies what FPGA image to program
- `fpga_image_info_alloc()` —Allocate an FPGA image info struct
- `fpga_image_info_free()` —Free an FPGA image info struct

int **fpga_region_program_fpga**(struct fpga_region * region)
program FPGA

Parameters

struct fpga_region * region FPGA region

Description

Program an FPGA using fpga image info (region->info). If the region has a get_bridges function, the exclusive reference for the bridges will be held if programming succeeds. This is intended to prevent reprogramming the region until the caller considers it safe to do so. The caller will need to call `fpga_bridges_put()` before attempting to reprogram the region.

Return 0 for success or negative error code.

FPGA Manager flags

Flags used in the `fpga_image_info->flags` field

`FPGA_MGR_PARTIAL_RECONFIG`: do partial reconfiguration if supported

`FPGA_MGR_EXTERNAL_CONFIG`: FPGA has been configured prior to Linux booting

`FPGA_MGR_ENCRYPTED_BITSTREAM`: indicates bitstream is encrypted

`FPGA_MGR_BITSTREAM_LSB_FIRST`: SPI bitstream bit order is LSB first

`FPGA_MGR_COMPRESSED_BITSTREAM`: FPGA bitstream is compressed

struct **fpga_image_info**
information specific to a FPGA image

Definition

```
struct fpga_image_info {
  u32 flags;
  u32 enable_timeout_us;
  u32 disable_timeout_us;
```

(continues on next page)

(continued from previous page)

```
  u32 config_complete_timeout_us;
  char *firmware_name;
  struct sg_table *sgt;
  const char *buf;
  size_t count;
  int region_id;
  struct device *dev;
#ifdef CONFIG_OF;
  struct device_node *overlay;
#endif;
};
```

Members

flags boolean flags as defined above

enable_timeout_us maximum time to enable traffic through bridge (uSec)

disable_timeout_us maximum time to disable traffic through bridge (uSec)

config_complete_timeout_us maximum time for FPGA to switch to operating status in the write_complete op.

firmware_name name of FPGA image firmware file

sgt scatter/gather table containing FPGA image

buf contiguous buffer containing FPGA image

count size of buf

region_id id of target region

dev device that owns this

overlay Device Tree overlay

struct fpga_image_info * **fpga_image_info_alloc**(struct device * dev)
Allocate a FPGA image info struct

Parameters

struct device * dev owning device

Return

struct fpga_image_info or NULL

void **fpga_image_info_free**(struct fpga_image_info * info)
Free a FPGA image info struct

Parameters

struct fpga_image_info * info FPGA image info struct to free

CHAPTER

SIXTYONE

ACPI SUPPORT

61.1 Linuxized ACPICA - Introduction to ACPICA Release Automation

Author Lv Zheng <lv.zheng@intel.com>

61.1.1 Abstract

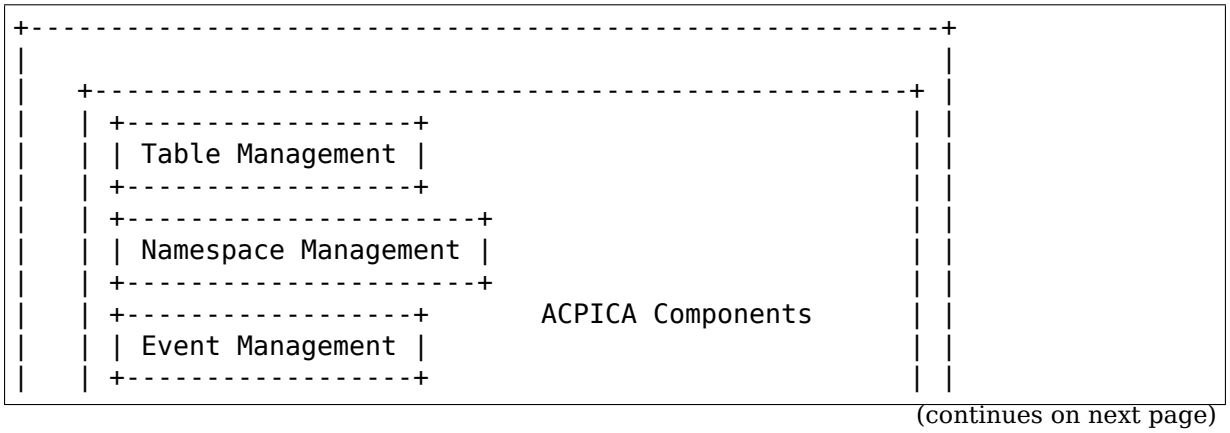
This document describes the ACPICA project and the relationship between ACPICA and Linux. It also describes how ACPICA code in drivers/acpi/acpica, include/acpi and tools/power/acpi is automatically updated to follow the upstream.

61.1.2 ACPICA Project

The ACPI Component Architecture (ACPICA) project provides an operating system (OS)-independent reference implementation of the Advanced Configuration and Power Interface Specification (ACPI). It has been adapted by various host OSes. By directly integrating ACPICA, Linux can also benefit from the application experiences of ACPICA from other host OSes.

The homepage of ACPICA project is: www.acpica.org, it is maintained and supported by Intel Corporation.

The following figure depicts the Linux ACPI subsystem where the ACPICA adaptation is included:



(continues on next page)

(continued from previous page)

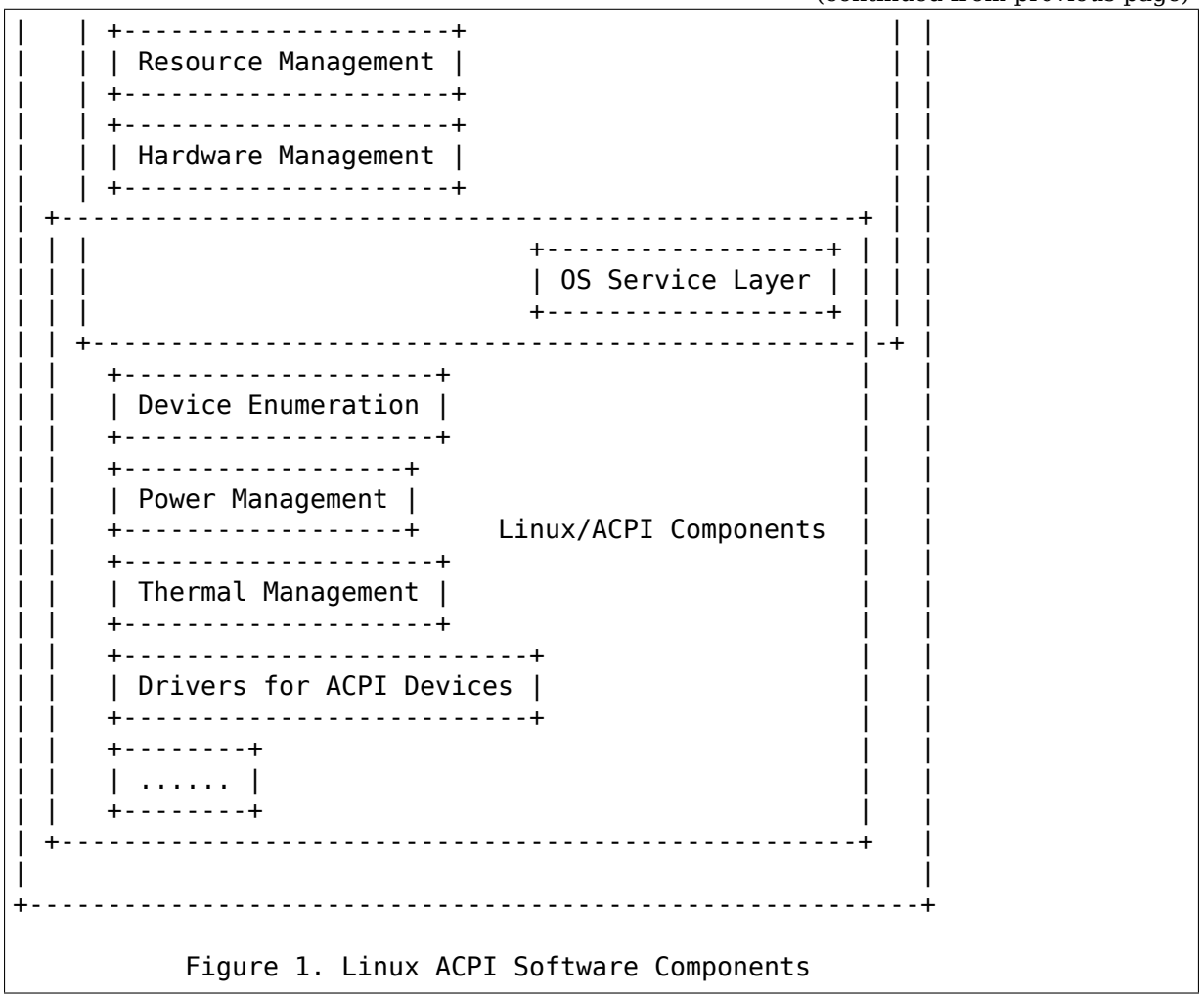


Figure 1. Linux ACPI Software Components

Note:

A. OS Service Layer - Provided by Linux to offer OS dependent implementation of the predefined ACPICA interfaces (acpi_os_*).

```
include/acpi/acpiosxf.h
drivers/acpi/osl.c
include/acpi/platform
include/asm/acenv.h
```

B. ACPICA Functionality - Released from ACPICA code base to offer OS independent implementation of the ACPICA interfaces (acpi_*).

```
drivers/acpi/acpica
include/acpi/ac*.h
tools/power/acpi
```

C. Linux/ACPI Functionality - Providing Linux specific ACPI functionality to the other Linux kernel subsystems and user space programs.

```
drivers/acpi
include/linux/acpi.h
```

(continues on next page)

(continued from previous page)

```
include/linux/acpi*.h
include/acpi
tools/power/acpi
```

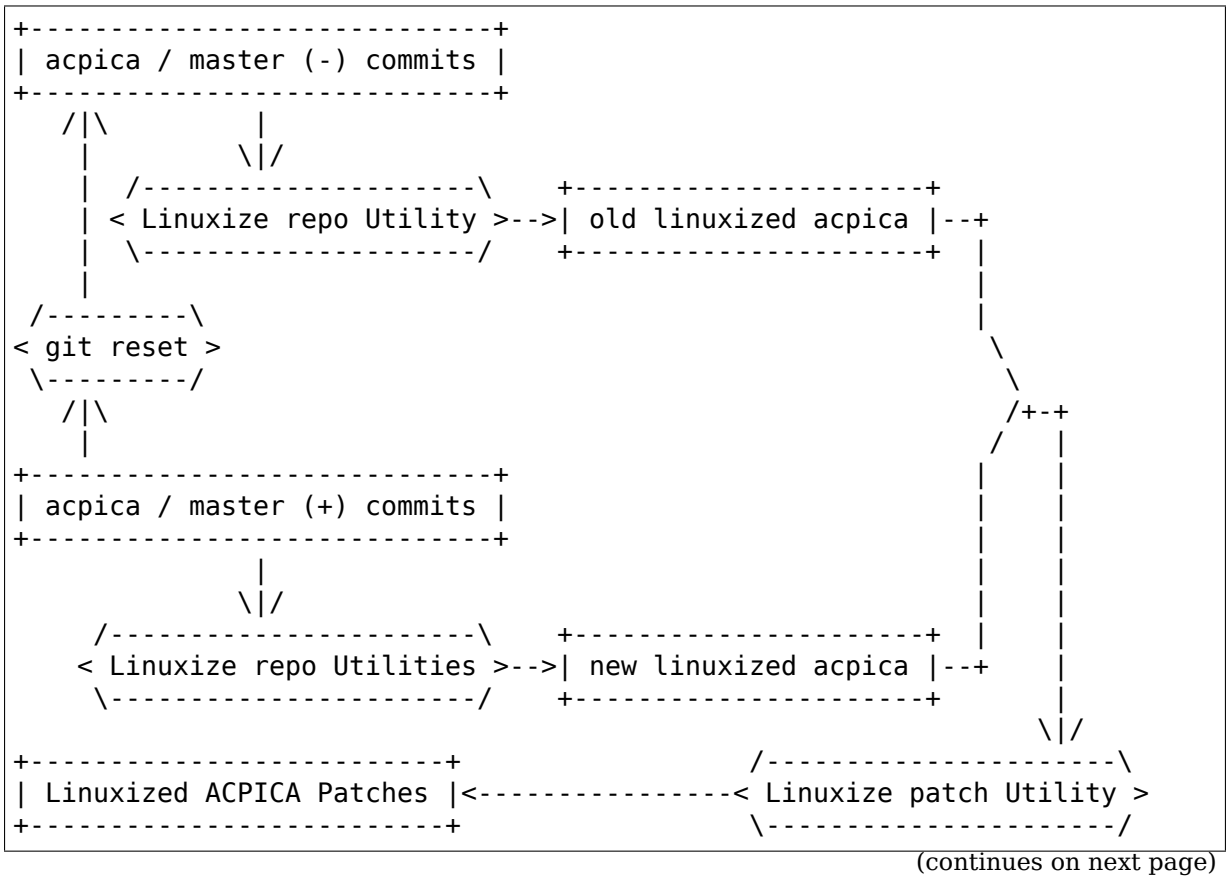
D. Architecture Specific ACPICA/ACPI Functionalities - Provided by the ACPI subsystem to offer architecture specific implementation of the ACPI interfaces. They are Linux specific components and are out of the scope of this document.

```
include/asm/acpi.h
include/asm/acpi*.h
arch/*/acpi
```

61.1.3 ACPICA Release

The ACPICA project maintains its code base at the following repository URL: https://github.com/acpica/acpica.git. As a rule, a release is made every month.

As the coding style adopted by the ACPICA project is not acceptable by Linux, there is a release process to convert the ACPICA git commits into Linux patches. The patches generated by this process are referred to as "linuxized ACPICA patches". The release process is carried out on a local copy the ACPICA git repository. Each commit in the monthly release is converted into a linuxized ACPICA patch. Together, they form the monthly ACPICA release patchset for the Linux ACPI community. This process is illustrated in the following figure:



(continues on next page)

(continued from previous page)

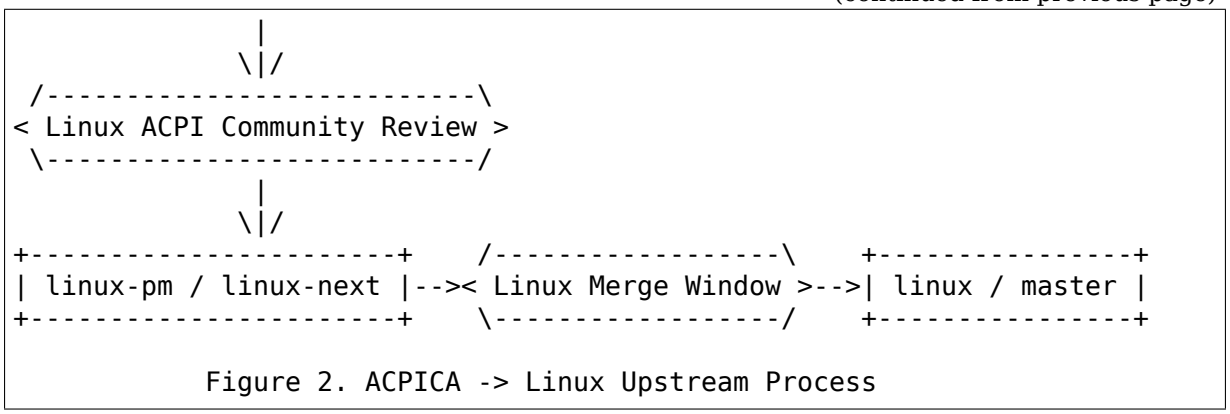


Figure 2. ACPICA -> Linux Upstream Process

Note:

A. Linuxize Utilities - Provided by the ACPICA repository, including a utility located in source/tools/acpisrc folder and a number of scripts located in generate/linux folder.

B. acpica / master - “master” branch of the git repository at <https://github.com/acpica/acpica.git>.

C. linux-pm / linux-next - “linux-next” branch of the git repository at <https://git.kernel.org/pub/scm/linux/kernel/git/rafael/linux-pm.git>.

D. linux / master - “master” branch of the git repository at <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git>.

Before the linuxized ACPICA patches are sent to the Linux ACPI community for review, there is a quality assurance build test process to reduce porting issues. Currently this build process only takes care of the following kernel configuration options: CONFIG_ACPI/CONFIG_ACPI_DEBUG/CONFIG_ACPI_DEBUGGER

61.1.4 ACPICA Divergences

Ideally, all of the ACPICA commits should be converted into Linux patches automatically without manual modifications, the “linux / master” tree should contain the ACPICA code that exactly corresponds to the ACPICA code contained in “new linuxized acpica” tree and it should be possible to run the release process fully automatically.

As a matter of fact, however, there are source code differences between the ACPICA code in Linux and the upstream ACPICA code, referred to as “ACPICA Divergences”.

The various sources of ACPICA divergences include:

1. Legacy divergences - Before the current ACPICA release process was established, there already had been divergences between Linux and ACPICA. Over the past several years those divergences have been greatly reduced, but there still are several ones and it takes time to figure out the underlying reasons for their existence.

2. Manual modifications - Any manual modification (eg. coding style fixes) made directly in the Linux sources obviously hurts the ACPICA release automation. Thus it is recommended to fix such issues in the ACPICA upstream source code and generate the linuxized fix using the ACPICA release utilities (please refer to Section 4 below for the details).

3. Linux specific features - Sometimes it’s impossible to use the current ACPICA APIs to implement features required by the Linux kernel, so Linux developers occasionally have to change ACPICA code directly. Those changes may not be acceptable by ACPICA upstream and in such cases they are left as committed ACPICA divergences unless the ACPICA side can implement new mechanisms as replacements for them.

4. ACPICA release fixups - ACPICA only tests commits using a set of the user space simulation utilities, thus the linuxized ACPICA patches may break the Linux kernel, leaving us build/boot failures. In order to avoid breaking Linux bisection, fixes are applied directly to the linuxized ACPICA patches during the release process. When the release fixups are backported to the upstream ACPICA sources, they must follow the upstream ACPICA rules and so further modifications may appear. That may result in the appearance of new divergences.

5. Fast tracking of ACPICA commits - Some ACPICA commits are regression fixes or stable-candidate material, so they are applied in advance with respect to the ACPICA release process. If such commits are reverted or rebased on the ACPICA side in order to offer better solutions, new ACPICA divergences are generated.

61.1.5 ACPICA Development

This paragraph guides Linux developers to use the ACPICA upstream release utilities to obtain Linux patches corresponding to upstream ACPICA commits before they become available from the ACPICA release process.

1. Cherry-pick an ACPICA commit

First you need to git clone the ACPICA repository and the ACPICA change you want to cherry pick must be committed into the local repository.

Then the gen-patch.sh command can help to cherry-pick an ACPICA commit from the ACPICA local repository:

```
$ git clone https://github.com/acpica/acpica
$ cd acpica
$ generate/linux/gen-patch.sh -u [commit ID]
```

Here the commit ID is the ACPICA local repository commit ID you want to cherry pick. It can be omitted if the commit is “HEAD”.

2. Cherry-pick recent ACPICA commits

Sometimes you need to rebase your code on top of the most recent ACPICA changes that haven’t been applied to Linux yet.

You can generate the ACPICA release series yourself and rebase your code on top of the generated ACPICA release patches:

```
$ git clone https://github.com/acpica/acpica
$ cd acpica
$ generate/linux/make-patches.sh -u [commit ID]
```

The commit ID should be the last ACPICA commit accepted by Linux. Usually, it is the commit modifying ACPI_CA_VERSION. It can be found by executing "git blame source/include/acpixf.h" and referencing the line that contains "ACPI_CA_VERSION".

3. Inspect the current divergences

If you have local copies of both Linux and upstream ACPICA, you can generate a diff file indicating the state of the current divergences:

```
# git clone https://github.com/acpica/acpica
# git clone https://git.kernel.org/pub/scm/linux/kernel/git/
↪torvalds/linux.git
# cd acpica
# generate/linux/divergences.sh -s ../linux
```

61.2 ACPI Scan Handlers

Author Rafael J. Wysocki <rafael.j.wysocki@intel.com>

During system initialization and ACPI-based device hot-add, the ACPI namespace is scanned in search of device objects that generally represent various pieces of hardware. This causes a struct acpi_device object to be created and registered with the driver core for every device object in the ACPI namespace and the hierarchy of those struct acpi_device objects reflects the namespace layout (i.e. parent device objects in the namespace are represented by parent struct acpi_device objects and analogously for their children). Those struct acpi_device objects are referred to as "device nodes" in what follows, but they should not be confused with struct device_node objects used by the Device Trees parsing code (although their role is analogous to the role of those objects).

During ACPI-based device hot-remove device nodes representing pieces of hardware being removed are unregistered and deleted.

The core ACPI namespace scanning code in drivers/acpi/scan.c carries out basic initialization of device nodes, such as retrieving common configuration information from the device objects represented by them and populating them with appropriate data, but some of them require additional handling after they have been registered. For example, if the given device node represents a PCI host bridge, its registration should cause the PCI bus under that bridge to be enumerated and PCI devices on that bus to be registered with the driver core. Similarly, if the device node represents a PCI interrupt link, it is necessary to configure that link so that the kernel can use it.

Those additional configuration tasks usually depend on the type of the hardware component represented by the given device node which can be determined on the basis of the device node's hardware ID (HID). They are performed by objects called ACPI scan handlers represented by the following structure:

```
struct acpi_scan_handler {
        const struct acpi_device_id *ids;
        struct list_head list_node;
        int (*attach)(struct acpi_device *dev, const struct acpi_device_id␣
↪*id);
        void (*detach)(struct acpi_device *dev);
};
```

where ids is the list of IDs of device nodes the given handler is supposed to take care of, list_node is the hook to the global list of ACPI scan handlers maintained by the ACPI core and the .attach() and .detach() callbacks are executed, respectively, after registration of new device nodes and before unregistration of device nodes the handler attached to previously.

The namespace scanning function, acpi_bus_scan(), first registers all of the device nodes in the given namespace scope with the driver core. Then, it tries to match a scan handler against each of them using the ids arrays of the available scan handlers. If a matching scan handler is found, its .attach() callback is executed for the given device node. If that callback returns 1, that means that the handler has claimed the device node and is now responsible for carrying out any additional configuration tasks related to it. It also will be responsible for preparing the device node for unregistration in that case. The device node's handler field is then populated with the address of the scan handler that has claimed it.

If the .attach() callback returns 0, it means that the device node is not interesting to the given scan handler and may be matched against the next scan handler in the list. If it returns a (negative) error code, that means that the namespace scan should be terminated due to a serious error. The error code returned should then reflect the type of the error.

The namespace trimming function, acpi_bus_trim(), first executes .detach() callbacks from the scan handlers of all device nodes in the given namespace scope (if they have scan handlers). Next, it unregisters all of the device nodes in that scope.

ACPI scan handlers can be added to the list maintained by the ACPI core with the help of the acpi_scan_add_handler() function taking a pointer to the new scan handler as an argument. The order in which scan handlers are added to the list is the order in which they are matched against device nodes during namespace scans.

All scan handles must be added to the list before acpi_bus_scan() is run for the first time and they cannot be removed from it.

CHAPTER

SIXTYTWO

KERNEL DRIVER LP855X

Backlight driver for LP855x ICs

Supported chips:

> Texas Instruments LP8550, LP8551, LP8552, LP8553, LP8555, LP8556 and LP8557

Author: Milo(Woogyom) Kim <milo.kim@ti.com>

62.1 Description

- Brightness control

  Brightness can be controlled by the pwm input or the i2c command. The lp855x driver supports both cases.

- Device attributes

  1) bl_ctl_mode

  Backlight control mode.

  Value: pwm based or register based

  2) chip_id

  The lp855x chip id.

  Value: lp8550/lp8551/lp8552/lp8553/lp8555/lp8556/lp8557

62.2 Platform data for lp855x

For supporting platform specific data, the lp855x platform data can be used.

- **name:** Backlight driver name. If it is not defined, default name is set.
- **device_control:** Value of DEVICE CONTROL register.
- **initial_brightness:** Initial value of backlight brightness.
- **period_ns:** Platform specific PWM period value. unit is nano. Only valid when brightness is pwm input mode.
- **size_program:** Total size of lp855x_rom_data.

- **rom_data:** List of new eeprom/eprom registers.

62.2.1 Examples

1) lp8552 platform data: i2c register mode with new eeprom data:

```
#define EEPROM_A5_ADDR      0xA5
#define EEPROM_A5_VAL       0x4f    /* EN_VSYNC=0 */

static struct lp855x_rom_data lp8552_eeprom_arr[] = {
    {EEPROM_A5_ADDR, EEPROM_A5_VAL},
};

static struct lp855x_platform_data lp8552_pdata = {
    .name = "lcd-bl",
    .device_control = I2C_CONFIG(LP8552),
    .initial_brightness = INITIAL_BRT,
    .size_program = ARRAY_SIZE(lp8552_eeprom_arr),
    .rom_data = lp8552_eeprom_arr,
};
```

2) lp8556 platform data: pwm input mode with default rom data:

```
static struct lp855x_platform_data lp8556_pdata = {
    .device_control = PWM_CONFIG(LP8556),
    .initial_brightness = INITIAL_BRT,
    .period_ns = 1000000,
};
```

CHAPTER

SIXTYTHREE

KERNEL CONNECTOR

Kernel connector - new netlink based userspace <-> kernel space easy to use communication module.

The Connector driver makes it easy to connect various agents using a netlink based network. One must register a callback and an identifier. When the driver receives a special netlink message with the appropriate identifier, the appropriate callback will be called.

From the userspace point of view it's quite straightforward:

- socket();
- bind();
- send();
- recv();

But if kernelspace wants to use the full power of such connections, the driver writer must create special sockets, must know about struct sk_buff handling, etc ···The Connector driver allows any kernelspace agents to use netlink based networking for inter-process communication in a significantly easier way:

```
int cn_add_callback(struct cb_id *id, char *name, void (*callback) (struct␣
↪cn_msg *, struct netlink_skb_parms *));
void cn_netlink_send_multi(struct cn_msg *msg, u16 len, u32 portid, u32 __
↪group, int gfp_mask);
void cn_netlink_send(struct cn_msg *msg, u32 portid, u32 __group, int gfp_
↪mask);

struct cb_id
{
      __u32                   idx;
      __u32                   val;
};
```

idx and val are unique identifiers which must be registered in the connector.h header for in-kernel usage. void (*callback) (void *) is a callback function which will be called when a message with above idx.val is received by the connector core. The argument for that function must be dereferenced to struct cn_msg *:

```
struct cn_msg
{
      struct cb_id            id;
```

(continues on next page)

(continued from previous page)

```
        __u32                   seq;
        __u32                   ack;

        __u32                   len;    /* Length of the following data */
        __u8                    data[0];
};
```

63.1 Connector interfaces

int **cn_add_callback**(struct cb_id * id, const char * name, void (*callback)(struct cn_msg *, struct netlink_skb_parms *))
Registers new callback with connector core.

Parameters

struct cb_id * id unique connector's user identifier. It must be registered in connector.h for legal in-kernel users.

const char * name connector's callback symbolic name.

void (*)(struct cn_msg *, struct netlink_skb_parms *) callback connector's callback. parameters are cn_msg and the sender's credentials

void **cn_del_callback**(struct cb_id * id)
Unregisters new callback with connector core.

Parameters

struct cb_id * id unique connector's user identifier.

int **cn_netlink_send_mult**(struct cn_msg * msg, u16 len, u32 portid, u32 group, gfp_t gfp_mask)
Sends message to the specified groups.

Parameters

struct cn_msg * msg message header(with attached data).

u16 len Number of **msg** to be sent.

u32 portid destination port. If non-zero the message will be sent to the given port, which should be set to the original sender.

u32 group destination group. If **portid** and **group** is zero, then appropriate group will be searched through all registered connector users, and message will be delivered to the group which was created for user with the same ID as in **msg**. If **group** is not zero, then message will be delivered to the specified group.

gfp_t gfp_mask GFP mask.

Description

It can be safely called from softirq context, but may silently fail under strong memory pressure.

If there are no listeners for given group -ESRCH can be returned.

int **cn_netlink_send**(struct cn_msg * msg, u32 portid, u32 group, gfp_t gfp_mask)
Sends message to the specified groups.

Parameters

struct cn_msg * msg message header(with attached data).

u32 portid destination port. If non-zero the message will be sent to the given port, which should be set to the original sender.

u32 group destination group. If **portid** and **group** is zero, then appropriate group will be searched through all registered connector users, and message will be delivered to the group which was created for user with the same ID as in **msg**. If **group** is not zero, then message will be delivered to the specified group.

gfp_t gfp_mask GFP mask.

Description

It can be safely called from softirq context, but may silently fail under strong memory pressure.

If there are no listeners for given group -ESRCH can be returned.

Note: When registering new callback user, connector core assigns netlink group to the user which is equal to its id.idx.

63.2 Protocol description

The current framework offers a transport layer with fixed headers. The recommended protocol which uses such a header is as following:

msg->seq and msg->ack are used to determine message genealogy. When someone sends a message, they use a locally unique sequence and random acknowledge number. The sequence number may be copied into nlmsghdr->nlmsg_seq too.

The sequence number is incremented with each message sent.

If you expect a reply to the message, then the sequence number in the received message MUST be the same as in the original message, and the acknowledge number MUST be the same + 1.

If we receive a message and its sequence number is not equal to one we are expecting, then it is a new message. If we receive a message and its sequence number is the same as one we are expecting, but its acknowledge is not equal to the sequence number in the original message + 1, then it is a new message.

Obviously, the protocol header contains the above id.

The connector allows event notification in the following form: kernel driver or userspace process can ask connector to notify it when selected ids will be turned

on or off (registered or unregistered its callback). It is done by sending a special command to the connector driver (it also registers itself with id={-1, -1}).

As example of this usage can be found in the cn_test.c module which uses the connector to request notification and to send messages.

63.3 Reliability

Netlink itself is not a reliable protocol. That means that messages can be lost due to memory pressure or process' receiving queue overflowed, so caller is warned that it must be prepared. That is why the struct cn_msg [main connector's message header] contains u32 seq and u32 ack fields.

63.4 Userspace usage

2.6.14 has a new netlink socket implementation, which by default does not allow people to send data to netlink groups other than 1. So, if you wish to use a netlink socket (for example using connector) with a different group number, the userspace application must subscribe to that group first. It can be achieved by the following pseudocode:

```
s = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_CONNECTOR);

l_local.nl_family = AF_NETLINK;
l_local.nl_groups = 12345;
l_local.nl_pid = 0;

if (bind(s, (struct sockaddr *)&l_local, sizeof(struct sockaddr_nl)) == -
↪1) {
      perror("bind");
      close(s);
      return -1;
}

{
      int on = l_local.nl_groups;
      setsockopt(s, 270, 1, &on, sizeof(on));
}
```

Where 270 above is SOL_NETLINK, and 1 is a NETLINK_ADD_MEMBERSHIP socket option. To drop a multicast subscription, one should call the above socket option with the NETLINK_DROP_MEMBERSHIP parameter which is defined as 0.

2.6.14 netlink code only allows to select a group which is less or equal to the maximum group number, which is used at netlink_kernel_create() time. In case of connector it is CN_NETLINK_USERS + 0xf, so if you want to use group number 12345, you must increment CN_NETLINK_USERS to that number. Additional 0xf numbers are allocated to be used by non-in-kernel users.

Due to this limitation, group 0xffffffff does not work now, so one can not use add/remove connector's group notifications, but as far as I know, only cn_test.c test module used it.

Some work in netlink area is still being done, so things can be changed in 2.6.15 timeframe, if it will happen, documentation will be updated for that kernel.

63.5 Code samples

Sample code for a connector test module and user space can be found in samples/connector/. To build this code, enable CONFIG_CONNECTOR and CONFIG_SAMPLES.

CHAPTER

SIXTYFOUR

CONSOLE DRIVERS

The Linux kernel has 2 general types of console drivers. The first type is assigned by the kernel to all the virtual consoles during the boot process. This type will be called ‘system driver’, and only one system driver is allowed to exist. The system driver is persistent and it can never be unloaded, though it may become inactive.

The second type has to be explicitly loaded and unloaded. This will be called ‘modular driver’ by this document. Multiple modular drivers can coexist at any time with each driver sharing the console with other drivers including the system driver. However, modular drivers cannot take over the console that is currently occupied by another modular driver. (Exception: Drivers that call do_take_over_console() will succeed in the takeover regardless of the type of driver occupying the consoles.) They can only take over the console that is occupied by the system driver. In the same token, if the modular driver is released by the console, the system driver will take over.

Modular drivers, from the programmer’s point of view, have to call:

```
do_take_over_console() - load and bind driver to console layer
give_up_console() - unload driver; it will only work if driver
                    is fully unbound
```

In newer kernels, the following are also available:

```
do_register_con_driver()
do_unregister_con_driver()
```

If sysfs is enabled, the contents of /sys/class/vtconsole can be examined. This shows the console backends currently registered by the system which are named vtcon<n> where <n> is an integer from 0 to 15. Thus:

```
ls /sys/class/vtconsole
.  ..  vtcon0  vtcon1
```

Each directory in /sys/class/vtconsole has 3 files:

```
ls /sys/class/vtconsole/vtcon0
.  ..  bind  name  uevent
```

What do these files signify?

1. bind - this is a read/write file. It shows the status of the driver if read, or acts to bind or unbind the driver to the virtual consoles when written to. The possible values are:

0

- means the driver is not bound and if echo'ed, commands the driver to unbind

1

- means the driver is bound and if echo'ed, commands the driver to bind

2. name - read-only file. Shows the name of the driver in this format:

```
cat /sys/class/vtconsole/vtcon0/name
(S) VGA+

    '(S)' stands for a (S)ystem driver, i.e., it cannot be directly
    commanded to bind or unbind

    'VGA+' is the name of the driver

cat /sys/class/vtconsole/vtcon1/name
(M) frame buffer device

    In this case, '(M)' stands for a (M)odular driver, one that can be
    directly commanded to bind or unbind.
```

3. uevent - ignore this file

When unbinding, the modular driver is detached first, and then the system driver takes over the consoles vacated by the driver. Binding, on the other hand, will bind the driver to the consoles that are currently occupied by a system driver.

NOTE1: Binding and unbinding must be selected in Kconfig. It's under:

```
Device Drivers ->
    Character devices ->
            Support for binding and unbinding console drivers
```

NOTE2: If any of the virtual consoles are in KD_GRAPHICS mode, then binding or unbinding will not succeed. An example of an application that sets the console to KD_GRAPHICS is X.

How useful is this feature? This is very useful for console driver developers. By unbinding the driver from the console layer, one can unload the driver, make changes, recompile, reload and rebind the driver without any need for rebooting the kernel. For regular users who may want to switch from framebuffer console to VGA console and vice versa, this feature also makes this possible. (NOTE NOTE NOTE: Please read fbcon.txt under Documentation/fb for more details.)

64.1 Notes for developers

do_take_over_console() is now broken up into:

```
do_register_con_driver()
do_bind_con_driver() - private function
```

give_up_console() is a wrapper to do_unregister_con_driver(), and a driver must be fully unbound for this call to succeed. con_is_bound() will check if the driver is bound or not.

64.2 Guidelines for console driver writers

In order for binding to and unbinding from the console to properly work, console drivers must follow these guidelines:

1. All drivers, except system drivers, must call either do_register_con_driver() or do_take_over_console(). do_register_con_driver() will just add the driver to the console's internal list. It won't take over the console. do_take_over_console(), as it name implies, will also take over (or bind to) the console.
2. All resources allocated during con->con_init() must be released in con->con_deinit().
3. All resources allocated in con->con_startup() must be released when the driver, which was previously bound, becomes unbound. The console layer does not have a complementary call to con->con_startup() so it's up to the driver to check when it's legal to release these resources. Calling con_is_bound() in con->con_deinit() will help. If the call returned false(), then it's safe to release the resources. This balance has to be ensured because con->con_startup() can be called again when a request to rebind the driver to the console arrives.
4. Upon exit of the driver, ensure that the driver is totally unbound. If the condition is satisfied, then the driver must call do_unregister_con_driver() or give_up_console().
5. do_unregister_con_driver() can also be called on conditions which make it impossible for the driver to service console requests. This can happen with the framebuffer console that suddenly lost all of its drivers.

The current crop of console drivers should still work correctly, but binding and unbinding them may cause problems. With minimal fixes, these drivers can be made to work correctly.

Antonino Daplas <adaplas@pol.net>

CHAPTER

SIXTYFIVE

DELL SYSTEMS MANAGEMENT BASE DRIVER

65.1 Overview

The Dell Systems Management Base Driver provides a sysfs interface for systems management software such as Dell OpenManage to perform system management interrupts and host control actions (system power cycle or power off after OS shutdown) on certain Dell systems.

Dell OpenManage requires this driver on the following Dell PowerEdge systems: 300, 1300, 1400, 400SC, 500SC, 1500SC, 1550, 600SC, 1600SC, 650, 1655MC, 700, and 750. Other Dell software such as the open source libsmbios project is expected to make use of this driver, and it may include the use of this driver on other Dell systems.

The Dell libsmbios project aims towards providing access to as much BIOS information as possible. See http://linux.dell.com/libsmbios/main/ for more information about the libsmbios project.

65.2 System Management Interrupt

On some Dell systems, systems management software must access certain management information via a system management interrupt (SMI). The SMI data buffer must reside in 32-bit address space, and the physical address of the buffer is required for the SMI. The driver maintains the memory required for the SMI and provides a way for the application to generate the SMI. The driver creates the following sysfs entries for systems management software to perform these system management interrupts:

```
/sys/devices/platform/dcdbas/smi_data
/sys/devices/platform/dcdbas/smi_data_buf_phys_addr
/sys/devices/platform/dcdbas/smi_data_buf_size
/sys/devices/platform/dcdbas/smi_request
```

Systems management software must perform the following steps to execute a SMI using this driver:

1) Lock smi_data.

2) Write system management command to smi_data.

3) Write “1” to smi_request to generate a calling interface SMI or “2” to generate a raw SMI.

4) Read system management command response from smi_data.

5) Unlock smi_data.

65.3 Host Control Action

Dell OpenManage supports a host control feature that allows the administrator to perform a power cycle or power off of the system after the OS has finished shutting down. On some Dell systems, this host control feature requires that a driver perform a SMI after the OS has finished shutting down.

The driver creates the following sysfs entries for systems management software to schedule the driver to perform a power cycle or power off host control action after the system has finished shutting down:

/sys/devices/platform/dcdbas/host_control_action /sys/devices/platform/dcdbas/host_control_sm
/sys/devices/platform/dcdbas/host_control_on_shutdown

Dell OpenManage performs the following steps to execute a power cycle or power off host control action using this driver:

1) Write host control action to be performed to host_control_action.

2) Write type of SMI that driver needs to perform to host_control_smi_type.

3) Write “1” to host_control_on_shutdown to enable host control action.

4) Initiate OS shutdown. (Driver will perform host control SMI when it is notified that the OS has finished shutting down.)

65.4 Host Control SMI Type

The following table shows the value to write to host_control_smi_type to perform a power cycle or power off host control action:

PowerEdge System	Host Control SMI Type
300	HC_SMITYPE_TYPE1
1300	HC_SMITYPE_TYPE1
1400	HC_SMITYPE_TYPE2
500SC	HC_SMITYPE_TYPE2
1500SC	HC_SMITYPE_TYPE2
1550	HC_SMITYPE_TYPE2
600SC	HC_SMITYPE_TYPE2
1600SC	HC_SMITYPE_TYPE2
650	HC_SMITYPE_TYPE2
1655MC	HC_SMITYPE_TYPE2
700	HC_SMITYPE_TYPE3
750	HC_SMITYPE_TYPE3

CHAPTER

SIXTYSIX

EISA BUS SUPPORT

Author Marc Zyngier <maz@wild-wind.fr.eu.org>

This document groups random notes about porting EISA drivers to the new EISA/sysfs API.

Starting from version 2.5.59, the EISA bus is almost given the same status as other much more mainstream busses such as PCI or USB. This has been possible through sysfs, which defines a nice enough set of abstractions to manage busses, devices and drivers.

Although the new API is quite simple to use, converting existing drivers to the new infrastructure is not an easy task (mostly because detection code is generally also used to probe ISA cards). Moreover, most EISA drivers are among the oldest Linux drivers so, as you can imagine, some dust has settled here over the years.

The EISA infrastructure is made up of three parts:

- The bus code implements most of the generic code. It is shared among all the architectures that the EISA code runs on. It implements bus probing (detecting EISA cards available on the bus), allocates I/O resources, allows fancy naming through sysfs, and offers interfaces for driver to register.
- The bus root driver implements the glue between the bus hardware and the generic bus code. It is responsible for discovering the device implementing the bus, and setting it up to be latter probed by the bus code. This can go from something as simple as reserving an I/O region on x86, to the rather more complex, like the hppa EISA code. This is the part to implement in order to have EISA running on an "new" platform.
- The driver offers the bus a list of devices that it manages, and implements the necessary callbacks to probe and release devices whenever told to.

Every function/structure below lives in <linux/eisa.h>, which depends heavily on <linux/device.h>.

66.1 Bus root driver

```
int eisa_root_register (struct eisa_root_device *root);
```

The eisa_root_register function is used to declare a device as the root of an EISA bus. The eisa_root_device structure holds a reference to this device, as well as some parameters for probing purposes:

```
struct eisa_root_device {
        struct device   *dev;    /* Pointer to bridge device */
        struct resource *res;
        unsigned long    bus_base_addr;
        int              slots;  /* Max slot number */
        int              force_probe; /* Probe even when no slot 0 */
        u64              dma_mask; /* from bridge device */
        int              bus_nr; /* Set by eisa_root_register */
        struct resource  eisa_root_res; /* ditto */
};
```

node	used for eisa_root_register internal purpose
dev	pointer to the root device
res	root device I/O resource
bus_base_addr	slot 0 address on this bus
slots	max slot number to probe
force_probe	Probe even when slot 0 is empty (no EISA mainboard)
dma_mask	Default DMA mask. Usually the bridge device dma_mask.
bus_nr	unique bus id, set by eisa_root_register

66.2 Driver

```
int eisa_driver_register (struct eisa_driver *edrv);
void eisa_driver_unregister (struct eisa_driver *edrv);
```

Clear enough ?

```
struct eisa_device_id {
        char sig[EISA_SIG_LEN];
        unsigned long driver_data;
};

struct eisa_driver {
        const struct eisa_device_id *id_table;
        struct device_driver         driver;
};
```

id_table	an array of NULL terminated EISA id strings, followed by an empty string. Each string can optionally be paired with a driver-dependent value (driver_data).
driver	a generic driver, such as described in Documentation/driver-api/driver-model/driver.rst. Only .name, .probe and .remove members are mandatory.

An example is the 3c59x driver:

```
static struct eisa_device_id vortex_eisa_ids[] = {
        { "TCM5920", EISA_3C592_OFFSET },
        { "TCM5970", EISA_3C597_OFFSET },
        { "" }
};

static struct eisa_driver vortex_eisa_driver = {
        .id_table = vortex_eisa_ids,
        .driver   = {
                .name    = "3c59x",
                .probe   = vortex_eisa_probe,
                .remove  = vortex_eisa_remove
        }
};
```

66.3 Device

The sysfs framework calls .probe and .remove functions upon device discovery and removal (note that the .remove function is only called when driver is built as a module).

Both functions are passed a pointer to a 'struct device', which is encapsulated in a 'struct eisa_device' described as follows:

```
struct eisa_device {
        struct eisa_device_id id;
        int                   slot;
        int                   state;
        unsigned long         base_addr;
        struct resource       res[EISA_MAX_RESOURCES];
        u64                   dma_mask;
        struct device         dev; /* generic device */
};
```

id	EISA id, as read from device. id.driver_data is set from the matching driver EISA id.
slot	slot number which the device was detected on
state	set of flags indicating the state of the device. Current flags are EISA_CONFIG_ENABLED and EISA_CONFIG_FORCED.
res	set of four 256 bytes I/O regions allocated to this device
dma_mask	DMA mask set from the parent device.
dev	generic device (see Documentation/driver-api/driver-model/device.rst)

You can get the ‘struct eisa_device’ from ‘struct device’ using the ‘to_eisa_device’ macro.

66.4 Misc stuff

```
void eisa_set_drvdata (struct eisa_device *edev, void *data);
```

Stores data into the device’s driver_data area.

```
void *eisa_get_drvdata (struct eisa_device *edev):
```

Gets the pointer previously stored into the device’s driver_data area.

```
int eisa_get_region_index (void *addr);
```

Returns the region number (0 <= x < EISA_MAX_RESOURCES) of a given address.

66.5 Kernel parameters

eisa_bus.enable_dev A comma-separated list of slots to be enabled, even if the firmware set the card as disabled. The driver must be able to properly initialize the device in such conditions.

eisa_bus.disable_dev A comma-separated list of slots to be enabled, even if the firmware set the card as enabled. The driver won’t be called to handle this device.

virtual_root.force_probe Force the probing code to probe EISA slots even when it cannot find an EISA compliant mainboard (nothing appears on slot 0). Defaults to 0 (don’t force), and set to 1 (force probing) when either CONFIG_ALPHA_JENSEN or CONFIG_EISA_VLB_PRIMING are set.

66.6 Random notes

Converting an EISA driver to the new API mostly involves deleting code (since probing is now in the core EISA code). Unfortunately, most drivers share their probing routine between ISA, and EISA. Special care must be taken when ripping out the EISA code, so other busses won’t suffer from these surgical strikes…

You must not expect any EISA device to be detected when returning from eisa_driver_register, since the chances are that the bus has not yet been probed. In fact, that’s what happens most of the time (the bus root driver usually kicks in rather late in the boot process). Unfortunately, most drivers are doing the probing by themselves, and expect to have explored the whole machine when they exit their probe routine.

For example, switching your favorite EISA SCSI card to the “hotplug” model is “the right thing” (tm).

66.7 Thanks

I'd like to thank the following people for their help:

- Xavier Benigni for lending me a wonderful Alpha Jensen,
- James Bottomley, Jeff Garzik for getting this stuff into the kernel,
- Andries Brouwer for contributing numerous EISA ids,
- Catrin Jones for coping with far too many machines at home.

CHAPTER

SIXTYSEVEN

ISA DRIVERS

The following text is adapted from the commit message of the initial commit of the ISA bus driver authored by Rene Herman.

During the recent "isa drivers using platform devices" discussion it was pointed out that (ALSA) ISA drivers ran into the problem of not having the option to fail driver load (device registration rather) upon not finding their hardware due to a probe() error not being passed up through the driver model. In the course of that, I suggested a separate ISA bus might be best; Russell King agreed and suggested this bus could use the .match() method for the actual device discovery.

The attached does this. For this old non (generically) discoverable ISA hardware only the driver itself can do discovery so as a difference with the platform_bus, this isa_bus also distributes match() up to the driver.

As another difference: these devices only exist in the driver model due to the driver creating them because it might want to drive them, meaning that all device creation has been made internal as well.

The usage model this provides is nice, and has been acked from the ALSA side by Takashi Iwai and Jaroslav Kysela. The ALSA driver module_init's now (for oldisa-only drivers) become:

```
static int __init alsa_card_foo_init(void)
{
        return isa_register_driver(&snd_foo_isa_driver, SNDRV_CARDS);
}

static void __exit alsa_card_foo_exit(void)
{
        isa_unregister_driver(&snd_foo_isa_driver);
}
```

Quite like the other bus models therefore. This removes a lot of duplicated init code from the ALSA ISA drivers.

The passed in isa_driver struct is the regular driver struct embedding a struct device_driver, the normal probe/remove/shutdown/suspend/resume callbacks, and as indicated that .match callback.

The "SNDRV_CARDS" you see being passed in is a "unsigned int ndev" parameter, indicating how many devices to create and call our methods with.

The platform_driver callbacks are called with a platform_device param; the isa_driver callbacks are being called with a `struct device *dev, unsigned int`

id pair directly – with the device creation completely internal to the bus it's much cleaner to not leak isa_dev's by passing them in at all. The id is the only thing we ever want other then the struct device anyways, and it makes for nicer code in the callbacks as well.

With this additional .match() callback ISA drivers have all options. If ALSA would want to keep the old non-load behaviour, it could stick all of the old .probe in .match, which would only keep them registered after everything was found to be present and accounted for. If it wanted the behaviour of always loading as it inadvertently did for a bit after the changeover to platform devices, it could just not provide a .match() and do everything in .probe() as before.

If it, as Takashi Iwai already suggested earlier as a way of following the model from saner buses more closely, wants to load when a later bind could conceivably succeed, it could use .match() for the prerequisites (such as checking the user wants the card enabled and that port/irq/dma values have been passed in) and .probe() for everything else. This is the nicest model.

To the code…

This exports only two functions; isa_{,un}register_driver().

isa_register_driver() register's the struct device_driver, and then loops over the passed in ndev creating devices and registering them. This causes the bus match method to be called for them, which is:

```
int isa_bus_match(struct device *dev, struct device_driver *driver)
{
        struct isa_driver *isa_driver = to_isa_driver(driver);

        if (dev->platform_data == isa_driver) {
                if (!isa_driver->match ||
                        isa_driver->match(dev, to_isa_dev(dev)->id))
                        return 1;
                dev->platform_data = NULL;
        }
        return 0;
}
```

The first thing this does is check if this device is in fact one of this driver's devices by seeing if the device's platform_data pointer is set to this driver. Platform devices compare strings, but we don't need to do that with everything being internal, so isa_register_driver() abuses dev->platform_data as a isa_driver pointer which we can then check here. I believe platform_data is available for this, but if rather not, moving the isa_driver pointer to the private struct isa_dev is ofcourse fine as well.

Then, if the the driver did not provide a .match, it matches. If it did, the driver match() method is called to determine a match.

If it did **not** match, dev->platform_data is reset to indicate this to isa_register_driver which can then unregister the device again.

If during all this, there's any error, or no devices matched at all everything is backed out again and the error, or -ENODEV, is returned.

isa_unregister_driver() just unregisters the matched devices and the driver itself.

module_isa_driver is a helper macro for ISA drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate code. Each module may only use this macro once, and calling it replaces module_init and module_exit.

max_num_isa_dev is a macro to determine the maximum possible number of ISA devices which may be registered in the I/O port address space given the address extent of the ISA devices.

CHAPTER

SIXTYEIGHT

ISA PLUG & PLAY SUPPORT BY JAROSLAV KYSELA <PEREX@SUSE.CZ>

68.1 Interface /proc/isapnp

The interface has been removed. See pnp.txt for more details.

68.2 Interface /proc/bus/isapnp

This directory allows access to ISA PnP cards and logical devices. The regular files contain the contents of ISA PnP registers for a logical device.

CHAPTER

SIXTYNINE

THE IO_MAPPING FUNCTIONS

69.1 API

The io_mapping functions in linux/io-mapping.h provide an abstraction for efficiently mapping small regions of an I/O device to the CPU. The initial usage is to support the large graphics aperture on 32-bit processors where ioremap_wc cannot be used to statically map the entire aperture to the CPU as it would consume too much of the kernel address space.

A mapping object is created during driver initialization using:

```
struct io_mapping *io_mapping_create_wc(unsigned long base,
                                        unsigned long size)
```

'base' is the bus address of the region to be made mappable, while 'size' indicates how large a mapping region to enable. Both are in bytes.

This _wc variant provides a mapping which may only be used with the io_mapping_map_atomic_wc or io_mapping_map_wc.

With this mapping object, individual pages can be mapped either atomically or not, depending on the necessary scheduling environment. Of course, atomic maps are more efficient:

```
void *io_mapping_map_atomic_wc(struct io_mapping *mapping,
                               unsigned long offset)
```

'offset'is the offset within the defined mapping region. Accessing addresses beyond the region specified in the creation function yields undefined results. Using an offset which is not page aligned yields an undefined result. The return value points to a single page in CPU address space.

This _wc variant returns a write-combining map to the page and may only be used with mappings created by io_mapping_create_wc

Note that the task may not sleep while holding this page mapped.

```
void io_mapping_unmap_atomic(void *vaddr)
```

'vaddr' must be the value returned by the last io_mapping_map_atomic_wc call. This unmaps the specified page and allows the task to sleep once again.

If you need to sleep while holding the lock, you can use the non-atomic variant, although they may be significantly slower.

```
void *io_mapping_map_wc(struct io_mapping *mapping,
                        unsigned long offset)
```

This works like io_mapping_map_atomic_wc except it allows the task to sleep while holding the page mapped.

```
void io_mapping_unmap(void *vaddr)
```

This works like io_mapping_unmap_atomic, except it is used for pages mapped with io_mapping_map_wc.

At driver close time, the io_mapping object must be freed:

```
void io_mapping_free(struct io_mapping *mapping)
```

69.2 Current Implementation

The initial implementation of these functions uses existing mapping mechanisms and so provides only an abstraction layer and no new functionality.

On 64-bit processors, io_mapping_create_wc calls ioremap_wc for the whole range, creating a permanent kernel-visible mapping to the resource. The map_atomic and map functions add the requested offset to the base of the virtual address returned by ioremap_wc.

On 32-bit processors with HIGHMEM defined, io_mapping_map_atomic_wc uses kmap_atomic_pfn to map the specified page in an atomic fashion; kmap_atomic_pfn isn't really supposed to be used with device pages, but it provides an efficient mapping for this usage.

On 32-bit processors without HIGHMEM defined, io_mapping_map_atomic_wc and io_mapping_map_wc both use ioremap_wc, a terribly inefficient function which performs an IPI to inform all processors about the new mapping. This results in a significant performance penalty.

CHAPTER

SEVENTY

ORDERING I/O WRITES TO MEMORY-MAPPED ADDRESSES

On some platforms, so-called memory-mapped I/O is weakly ordered. On such platforms, driver writers are responsible for ensuring that I/O writes to memory-mapped addresses on their device arrive in the order intended. This is typically done by reading a 'safe' device or bridge register, causing the I/O chipset to flush pending writes to the device before any reads are posted. A driver would usually use this technique immediately prior to the exit of a critical section of code protected by spinlocks. This would ensure that subsequent writes to I/O space arrived only after all prior writes (much like a memory barrier op, mb(), only with respect to I/O).

A more concrete example from a hypothetical device driver:

```
        ...
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  val = readl(my_status);
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  val = readl(my_status);
CPU B:  ...
CPU B:  writel(newval2, ring_ptr);
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
```

In the case above, the device may receive newval2 before it receives newval, which could cause problems. Fixing it is easy enough though:

```
        ...
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  val = readl(my_status);
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  (void)readl(safe_register); /* maybe a config register? */
CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  val = readl(my_status);
CPU B:  ...
CPU B:  writel(newval2, ring_ptr);
```

(continues on next page)

(continued from previous page)

```
CPU B:  (void)readl(safe_register); /* maybe a config register? */
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
```

Here, the reads from safe_register will cause the I/O chipset to flush any pending writes before actually posting the read to the chipset, preventing possible data corruption.

CHAPTER

SEVENTYONE

GENERIC COUNTER INTERFACE

71.1 Introduction

Counter devices are prevalent among a diverse spectrum of industries. The ubiquitous presence of these devices necessitates a common interface and standard of interaction and exposure. This driver API attempts to resolve the issue of duplicate code found among existing counter device drivers by introducing a generic counter interface for consumption. The Generic Counter interface enables drivers to support and expose a common set of components and functionality present in counter devices.

71.2 Theory

Counter devices can vary greatly in design, but regardless of whether some devices are quadrature encoder counters or tally counters, all counter devices consist of a core set of components. This core set of components, shared by all counter devices, is what forms the essence of the Generic Counter interface.

There are three core components to a counter:

- Signal: Stream of data to be evaluated by the counter.
- Synapse: Association of a Signal, and evaluation trigger, with a Count.
- Count: Accumulation of the effects of connected Synapses.

71.2.1 SIGNAL

A Signal represents a stream of data. This is the input data that is evaluated by the counter to determine the count data; e.g. a quadrature signal output line of a rotary encoder. Not all counter devices provide user access to the Signal data, so exposure is optional for drivers.

When the Signal data is available for user access, the Generic Counter interface provides the following available signal values:

- SIGNAL_LOW: Signal line is in a low state.
- SIGNAL_HIGH: Signal line is in a high state.

A Signal may be associated with one or more Counts.

71.2.2 SYNAPSE

A Synapse represents the association of a Signal with a Count. Signal data affects respective Count data, and the Synapse represents this relationship.

The Synapse action mode specifies the Signal data condition that triggers the respective Count's count function evaluation to update the count data. The Generic Counter interface provides the following available action modes:

- None: Signal does not trigger the count function. In Pulse-Direction count function mode, this Signal is evaluated as Direction.
- Rising Edge: Low state transitions to high state.
- Falling Edge: High state transitions to low state.
- Both Edges: Any state transition.

A counter is defined as a set of input signals associated with count data that are generated by the evaluation of the state of the associated input signals as defined by the respective count functions. Within the context of the Generic Counter interface, a counter consists of Counts each associated with a set of Signals, whose respective Synapse instances represent the count function update conditions for the associated Counts.

A Synapse associates one Signal with one Count.

71.2.3 COUNT

A Count represents the accumulation of the effects of connected Synapses; i.e. the count data for a set of Signals. The Generic Counter interface represents the count data as a natural number.

A Count has a count function mode which represents the update behavior for the count data. The Generic Counter interface provides the following available count function modes:

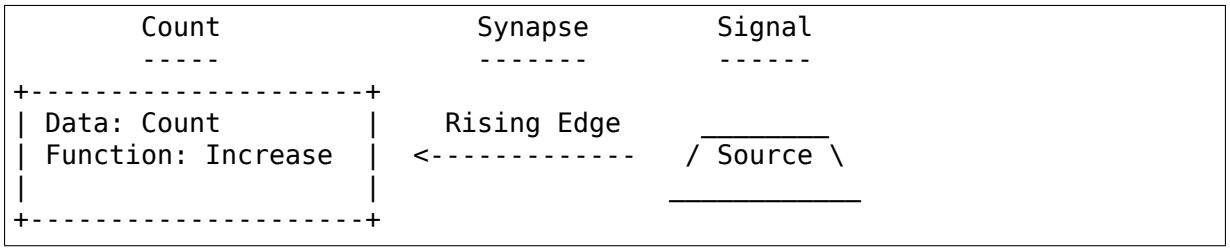
- Increase: Accumulated count is incremented.
- Decrease: Accumulated count is decremented.
- Pulse-Direction: Rising edges on signal A updates the respective count. The input level of signal B determines direction.
- Quadrature: A pair of quadrature encoding signals are evaluated to determine position and direction. The following Quadrature modes are available:
  - x1 A: If direction is forward, rising edges on quadrature pair signal A updates the respective count; if the direction is backward, falling edges on quadrature pair signal A updates the respective count. Quadrature encoding determines the direction.
  - x1 B: If direction is forward, rising edges on quadrature pair signal B updates the respective count; if the direction is backward, falling edges on quadrature pair signal B updates the respective count. Quadrature encoding determines the direction.

- x2 A: Any state transition on quadrature pair signal A updates the respective count. Quadrature encoding determines the direction.
- x2 B: Any state transition on quadrature pair signal B updates the respective count. Quadrature encoding determines the direction.
- x4: Any state transition on either quadrature pair signals updates the respective count. Quadrature encoding determines the direction.

A Count has a set of one or more associated Synapses.

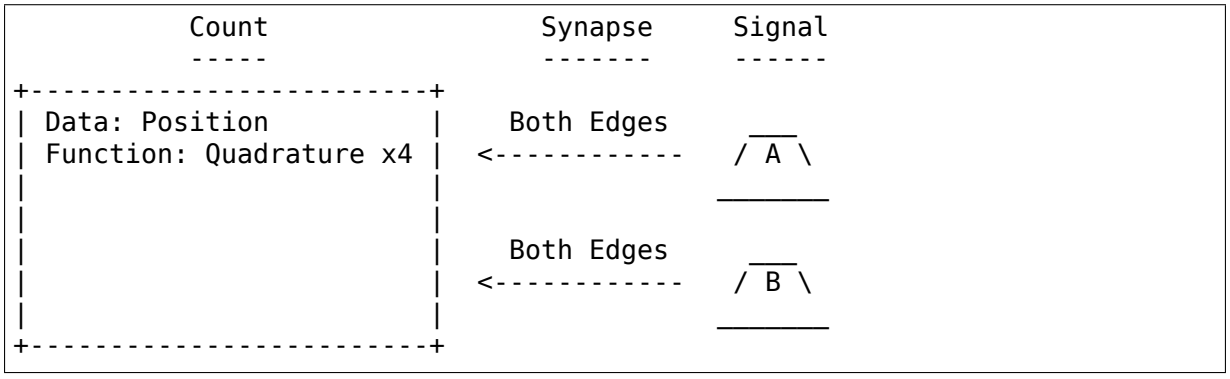
71.3 Paradigm

The most basic counter device may be expressed as a single Count associated with a single Signal via a single Synapse. Take for example a counter device which simply accumulates a count of rising edges on a source input line:



In this example, the Signal is a source input line with a pulsing voltage, while the Count is a persistent count value which is repeatedly incremented. The Signal is associated with the respective Count via a Synapse. The increase function is triggered by the Signal data condition specified by the Synapse – in this case a rising edge condition on the voltage input line. In summary, the counter device existence and behavior is aptly represented by respective Count, Signal, and Synapse components: a rising edge condition triggers an increase function on an accumulating count datum.

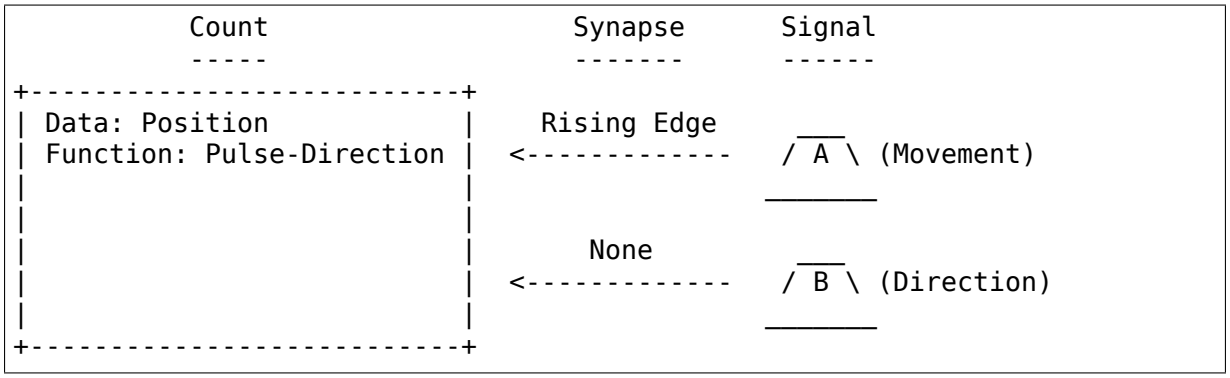
A counter device is not limited to a single Signal; in fact, in theory many Signals may be associated with even a single Count. For example, a quadrature encoder counter device can keep track of position based on the states of two input lines:



In this example, two Signals (quadrature encoder lines A and B) are associated with a single Count: a rising or falling edge on either A or B triggers the “Quadrature x4” function which determines the direction of movement and updates the respective position data. The “Quadrature x4” function is likely implemented in

the hardware of the quadrature encoder counter device; the Count, Signals, and Synapses simply represent this hardware behavior and functionality.

Signals associated with the same Count can have differing Synapse action mode conditions. For example, a quadrature encoder counter device operating in a non-quadrature Pulse-Direction mode could have one input line dedicated for movement and a second input line dedicated for direction:



Only Signal A triggers the “Pulse-Direction” update function, but the instantaneous state of Signal B is still required in order to know the direction so that the position data may be properly updated. Ultimately, both Signals are associated with the same Count via two respective Synapses, but only one Synapse has an active action mode condition which triggers the respective count function while the other is left with a “None” condition action mode to indicate its respective Signal’s availability for state evaluation despite its non-triggering mode.

Keep in mind that the Signal, Synapse, and Count are abstract representations which do not need to be closely married to their respective physical sources. This allows the user of a counter to divorce themselves from the nuances of physical components (such as whether an input line is differential or single-ended) and instead focus on the core idea of what the data and process represent (e.g. position as interpreted from quadrature encoding data).

71.4 Userspace Interface

Several sysfs attributes are generated by the Generic Counter interface, and reside under the /sys/bus/counter/devices/counterX directory, where counterX refers to the respective counter device. Please see Documentation/ABI/testing/sysfs-bus-counter for detailed information on each Generic Counter interface sysfs attribute.

Through these sysfs attributes, programs and scripts may interact with the Generic Counter paradigm Counts, Signals, and Synapses of respective counter devices.

71.5 Driver API

Driver authors may utilize the Generic Counter interface in their code by including the include/linux/counter.h header file. This header file provides several core data structures, function prototypes, and macros for defining a counter device.

struct **counter_signal_ext**
Counter Signal extensions

Definition

```
struct counter_signal_ext {
  const char *name;
  ssize_t (*read)(struct counter_device *counter, struct counter_signal␣
↪*signal, void *priv, char *buf);
  ssize_t (*write)(struct counter_device *counter,struct counter_signal␣
↪*signal, void *priv, const char *buf, size_t len);
  void *priv;
};
```

Members

name attribute name

read read callback for this attribute; may be NULL

write write callback for this attribute; may be NULL

priv data private to the driver

struct **counter_signal**
Counter Signal node

Definition

```
struct counter_signal {
  int id;
  const char *name;
  const struct counter_signal_ext *ext;
  size_t num_ext;
  void *priv;
};
```

Members

id unique ID used to identify signal

name device-specific Signal name; ideally, this should match the name as it appears in the datasheet documentation

ext optional array of Counter Signal extensions

num_ext number of Counter Signal extensions specified in **ext**

priv optional private data supplied by driver

struct **counter_signal_enum_ext**
Signal enum extension attribute

Definition

```
struct counter_signal_enum_ext {
  const char * const *items;
  size_t num_items;
  int (*get)(struct counter_device *counter, struct counter_signal *signal,
↪ size_t *item);
  int (*set)(struct counter_device *counter, struct counter_signal *signal,
↪ size_t item);
};
```

Members

items Array of strings

num_items Number of items specified in **items**

get Get callback function; may be NULL

set Set callback function; may be NULL

Description

The counter_signal_enum_ext structure can be used to implement enum style Signal extension attributes. Enum style attributes are those which have a set of strings that map to unsigned integer values. The Generic Counter Signal enum extension helper code takes care of mapping between value and string, as well as generating a "_available" file which contains a list of all available items. The get callback is used to query the currently active item; the index of the item within the respective items array is returned via the 'item' parameter. The set callback is called when the attribute is updated; the 'item' parameter contains the index of the newly activated item within the respective items array.

COUNTER_SIGNAL_ENUM(_name, _e)
Initialize Signal enum extension

Parameters

_name Attribute name

_e Pointer to a counter_signal_enum_ext structure

Description

This should usually be used together with `COUNTER_SIGNAL_ENUM_AVAILABLE()`

COUNTER_SIGNAL_ENUM_AVAILABLE(_name, _e)
Initialize Signal enum available extension

Parameters

_name Attribute name ("_available" will be appended to the name)

_e Pointer to a counter_signal_enum_ext structure

Description

Creates a read only attribute that lists all the available enum items in a newline separated list. This should usually be used together with `COUNTER_SIGNAL_ENUM()`

struct **counter_synapse**
Counter Synapse node

Definition

```
struct counter_synapse {
  size_t action;
  const enum counter_synapse_action *actions_list;
  size_t num_actions;
  struct counter_signal *signal;
};
```

Members

action index of current action mode

actions_list array of available action modes

num_actions number of action modes specified in **actions_list**

signal pointer to associated signal

struct **counter_count_ext**
Counter Count extension

Definition

```
struct counter_count_ext {
  const char *name;
  ssize_t (*read)(struct counter_device *counter, struct counter_count␣
↪*count, void *priv, char *buf);
  ssize_t (*write)(struct counter_device *counter,struct counter_count␣
↪*count, void *priv, const char *buf, size_t len);
  void *priv;
};
```

Members

name attribute name

read read callback for this attribute; may be NULL

write write callback for this attribute; may be NULL

priv data private to the driver

struct **counter_count**
Counter Count node

Definition

```
struct counter_count {
  int id;
  const char *name;
  size_t function;
  const enum counter_count_function *functions_list;
  size_t num_functions;
  struct counter_synapse *synapses;
  size_t num_synapses;
  const struct counter_count_ext *ext;
  size_t num_ext;
  void *priv;
};
```

Members

id unique ID used to identify Count

name device-specific Count name; ideally, this should match the name as it appears in the datasheet documentation

function index of current function mode

functions_list array available function modes

num_functions number of function modes specified in **functions_list**

synapses array of synapses for initialization

num_synapses number of synapses specified in **synapses**

ext optional array of Counter Count extensions

num_ext number of Counter Count extensions specified in **ext**

priv optional private data supplied by driver

struct **counter_count_enum_ext**
Count enum extension attribute

Definition

```
struct counter_count_enum_ext {
  const char * const *items;
  size_t num_items;
  int (*get)(struct counter_device *counter, struct counter_count *count,␣
↪size_t *item);
  int (*set)(struct counter_device *counter, struct counter_count *count,␣
↪size_t item);
};
```

Members

items Array of strings

num_items Number of items specified in **items**

get Get callback function; may be NULL

set Set callback function; may be NULL

Description

The counter_count_enum_ext structure can be used to implement enum style Count extension attributes. Enum style attributes are those which have a set of strings that map to unsigned integer values. The Generic Counter Count enum extension helper code takes care of mapping between value and string, as well as generating a "_available" file which contains a list of all available items. The get callback is used to query the currently active item; the index of the item within the respective items array is returned via the 'item' parameter. The set callback is called when the attribute is updated; the 'item' parameter contains the index of the newly activated item within the respective items array.

COUNTER_COUNT_ENUM(_name, _e)
Initialize Count enum extension

Parameters

_name Attribute name

_e Pointer to a counter_count_enum_ext structure

Description

This should usually be used together with `COUNTER_COUNT_ENUM_AVAILABLE()`

COUNTER_COUNT_ENUM_AVAILABLE(_name, _e)
Initialize Count enum available extension

Parameters

_name Attribute name (“_available” will be appended to the name)

_e Pointer to a counter_count_enum_ext structure

Description

Creates a read only attribute that lists all the available enum items in a newline separated list. This should usually be used together with `COUNTER_COUNT_ENUM()`

struct **counter_device_attr_group**
internal container for attribute group

Definition

```
struct counter_device_attr_group {
  struct attribute_group attr_group;
  struct list_head attr_list;
  size_t num_attr;
};
```

Members

attr_group Counter sysfs attributes group

attr_list list to keep track of created Counter sysfs attributes

num_attr number of Counter sysfs attributes

struct **counter_device_state**
internal state container for a Counter device

Definition

```
struct counter_device_state {
  int id;
  struct device dev;
  struct counter_device_attr_group *groups_list;
  size_t num_groups;
  const struct attribute_group **groups;
};
```

Members

id unique ID used to identify the Counter

dev internal device structure

groups_list attribute groups list (for Signals, Counts, and ext)

num_groups number of attribute groups containers

groups Counter sysfs attribute groups (to populate **dev.groups**)

struct **counter_ops**
Callbacks from driver

Definition

```
struct counter_ops {
  int (*signal_read)(struct counter_device *counter,struct counter_signal␣
↪*signal, enum counter_signal_value *val);
  int (*count_read)(struct counter_device *counter, struct counter_count␣
↪*count, unsigned long *val);
  int (*count_write)(struct counter_device *counter, struct counter_count␣
↪*count, unsigned long val);
  int (*function_get)(struct counter_device *counter, struct counter_count␣
↪*count, size_t *function);
  int (*function_set)(struct counter_device *counter, struct counter_count␣
↪*count, size_t function);
  int (*action_get)(struct counter_device *counter,struct counter_count␣
↪*count, struct counter_synapse *synapse, size_t *action);
  int (*action_set)(struct counter_device *counter,struct counter_count␣
↪*count, struct counter_synapse *synapse, size_t action);
};
```

Members

signal_read optional read callback for Signal attribute. The read value of the respective Signal should be passed back via the val parameter.

count_read optional read callback for Count attribute. The read value of the respective Count should be passed back via the val parameter.

count_write optional write callback for Count attribute. The write value for the respective Count is passed in via the val parameter.

function_get function to get the current count function mode. Returns 0 on success and negative error code on error. The index of the respective Count's returned function mode should be passed back via the function parameter.

function_set function to set the count function mode. function is the index of the requested function mode from the respective Count's functions_list array.

action_get function to get the current action mode. Returns 0 on success and negative error code on error. The index of the respective Synapse's returned action mode should be passed back via the action parameter.

action_set function to set the action mode. action is the index of the requested action mode from the respective Synapse's actions_list array.

struct **counter_device_ext**
Counter device extension

Definition

```
struct counter_device_ext {
  const char *name;
  ssize_t (*read)(struct counter_device *counter, void *priv, char *buf);
```

(continues on next page)

(continued from previous page)

```
  ssize_t (*write)(struct counter_device *counter, void *priv, const char␣
↪*buf, size_t len);
  void *priv;
};
```

Members

name attribute name

read read callback for this attribute; may be NULL

write write callback for this attribute; may be NULL

priv data private to the driver

struct **counter_device_enum_ext**
Counter enum extension attribute

Definition

```
struct counter_device_enum_ext {
  const char * const *items;
  size_t num_items;
  int (*get)(struct counter_device *counter, size_t *item);
  int (*set)(struct counter_device *counter, size_t item);
};
```

Members

items Array of strings

num_items Number of items specified in **items**

get Get callback function; may be NULL

set Set callback function; may be NULL

Description

The counter_device_enum_ext structure can be used to implement enum style Counter extension attributes. Enum style attributes are those which have a set of strings that map to unsigned integer values. The Generic Counter enum extension helper code takes care of mapping between value and string, as well as generating a "_available" file which contains a list of all available items. The get callback is used to query the currently active item; the index of the item within the respective items array is returned via the 'item' parameter. The set callback is called when the attribute is updated; the 'item' parameter contains the index of the newly activated item within the respective items array.

COUNTER_DEVICE_ENUM(_name, _e)
Initialize Counter enum extension

Parameters

_name Attribute name

_e Pointer to a counter_device_enum_ext structure

Description

This should usually be used together with COUNTER_DEVICE_ENUM_AVAILABLE()

COUNTER_DEVICE_ENUM_AVAILABLE(_name, _e)
Initialize Counter enum available extension

Parameters

_name Attribute name (“_available” will be appended to the name)

_e Pointer to a counter_device_enum_ext structure

Description

Creates a read only attribute that lists all the available enum items in a newline separated list. This should usually be used together with COUNTER_DEVICE_ENUM()

struct **counter_device**
Counter data structure

Definition

```
struct counter_device {
  const char *name;
  struct device *parent;
  struct counter_device_state *device_state;
  const struct counter_ops *ops;
  struct counter_signal *signals;
  size_t num_signals;
  struct counter_count *counts;
  size_t num_counts;
  const struct counter_device_ext *ext;
  size_t num_ext;
  void *priv;
};
```

Members

name name of the device as it appears in the datasheet

parent optional parent device providing the counters

device_state internal device state container

ops callbacks from driver

signals array of Signals

num_signals number of Signals specified in **signals**

counts array of Counts

num_counts number of Counts specified in **counts**

ext optional array of Counter device extensions

num_ext number of Counter device extensions specified in **ext**

priv optional private data supplied by driver

int **counter_register**(struct counter_device *const counter)
register Counter to the system

Parameters

struct counter_device *const counter pointer to Counter to register

Description

This function registers a Counter to the system. A sysfs “counter” directory will be created and populated with sysfs attributes correlating with the Counter Signals, Synapses, and Counts respectively.

void **counter_unregister**(struct counter_device *const counter)
unregister Counter from the system

Parameters

struct counter_device *const counter pointer to Counter to unregister

Description

The Counter is unregistered from the system; all allocated memory is freed.

int **devm_counter_register**(struct device * dev, struct counter_device *const counter)
Resource-managed counter_register

Parameters

struct device * dev device to allocate counter_device for

struct counter_device *const counter pointer to Counter to register

Description

Managed counter_register. The Counter registered with this function is automatically unregistered on driver detach. This function calls counter_register internally. Refer to that function for more information.

If an Counter registered with this function needs to be unregistered separately, devm_counter_unregister must be used.

Return

0 on success, negative error number on failure.

void **devm_counter_unregister**(struct device * dev, struct counter_device *const counter)
Resource-managed counter_unregister

Parameters

struct device * dev device this counter_device belongs to

struct counter_device *const counter pointer to Counter associated with the device

Description

Unregister Counter registered with devm_counter_register.

71.6 Implementation

To support a counter device, a driver must first allocate the available Counter Signals via counter_signal structures. These Signals should be stored as an array and set to the signals array member of an allocated counter_device structure before the Counter is registered to the system.

Counter Counts may be allocated via counter_count structures, and respective Counter Signal associations (Synapses) made via counter_synapse structures. Associated counter_synapse structures are stored as an array and set to the the synapses array member of the respective counter_count structure. These counter_count structures are set to the counts array member of an allocated counter_device structure before the Counter is registered to the system.

Driver callbacks should be provided to the counter_device structure via a constant counter_ops structure in order to communicate with the device: to read and write various Signals and Counts, and to set and get the "action mode" and "function mode" for various Synapses and Counts respectively.

A defined counter_device structure may be registered to the system by passing it to the counter_register function, and unregistered by passing it to the counter_unregister function. Similarly, the devm_counter_register and devm_counter_unregister functions may be used if device memory-managed registration is desired.

Extension sysfs attributes can be created for auxiliary functionality and data by passing in defined counter_device_ext, counter_count_ext, and counter_signal_ext structures. In these cases, the counter_device_ext structure is used for global/miscellaneous exposure and configuration of the respective Counter device, while the counter_count_ext and counter_signal_ext structures allow for auxiliary exposure and configuration of a specific Count or Signal respectively.

Determining the type of extension to create is a matter of scope.

- Signal extensions are attributes that expose information/control specific to a Signal. These types of attributes will exist under a Signal's directory in sysfs.

  For example, if you have an invert feature for a Signal, you can have a Signal extension called "invert" that toggles that feature: /sys/bus/counter/devices/counterX/signalY/invert

- Count extensions are attributes that expose information/control specific to a Count. These type of attributes will exist under a Count's directory in sysfs.

  For example, if you want to pause/unpause a Count from updating, you can have a Count extension called "enable" that toggles such: /sys/bus/counter/devices/counterX/countY/enable

- Device extensions are attributes that expose information/control non-specific to a particular Count or Signal. This is where you would put your global features or other miscellanous functionality.

  For example, if your device has an overtemp sensor, you can report the chip overheated via a device extension called "error_overtemp": /sys/bus/counter/devices/counterX/error_overtemp

71.7 Architecture

When the Generic Counter interface counter module is loaded, the counter_init function is called which registers a bus_type named “counter” to the system. Subsequently, when the module is unloaded, the counter_exit function is called which unregisters the bus_type named “counter” from the system.

Counter devices are registered to the system via the counter_register function, and later removed via the counter_unregister function. The counter_register function establishes a unique ID for the Counter device and creates a respective sysfs directory, where X is the mentioned unique ID:

/sys/bus/counter/devices/counterX

Sysfs attributes are created within the counterX directory to expose functionality, configurations, and data relating to the Counts, Signals, and Synapses of the Counter device, as well as options and information for the Counter device itself.

Each Signal has a directory created to house its relevant sysfs attributes, where Y is the unique ID of the respective Signal:

/sys/bus/counter/devices/counterX/signalY

Similarly, each Count has a directory created to house its relevant sysfs attributes, where Y is the unique ID of the respective Count:

/sys/bus/counter/devices/counterX/countY

For a more detailed breakdown of the available Generic Counter interface sysfs attributes, please refer to the Documentation/ABI/testing/sysfs-bus-counter file.

The Signals and Counts associated with the Counter device are registered to the system as well by the counter_register function. The signal_read/signal_write driver callbacks are associated with their respective Signal attributes, while the count_read/count_write and function_get/function_set driver callbacks are associated with their respective Count attributes; similarly, the same is true for the action_get/action_set driver callbacks and their respective Synapse attributes. If a driver callback is left undefined, then the respective read/write permission is left disabled for the relevant attributes.

Similarly, extension sysfs attributes are created for the defined counter_device_ext, counter_count_ext, and counter_signal_ext structures that are passed in.

CHAPTER

SEVENTYTWO

PBLK: PHYSICAL BLOCK DEVICE TARGET

pblk implements a fully associative, host-based FTL that exposes a traditional block I/O interface. Its primary responsibilities are:

- Map logical addresses onto physical addresses (4KB granularity) in a logical-to-physical (L2P) table.
- Maintain the integrity and consistency of the L2P table as well as its recovery from normal tear down and power outage.
- Deal with controller- and media-specific constrains.
- Handle I/O errors.
- Implement garbage collection.
- Maintain consistency across the I/O stack during synchronization points.

For more information please refer to:

http://lightnvm.io

which maintains updated FAQs, manual pages, technical documentation, tools, contacts, etc.

CHAPTER

SEVENTYTHREE

MEMORY CONTROLLER DRIVERS

73.1 TI EMIF SDRAM Controller Driver

73.1.1 Author

Aneesh V <aneesh@ti.com>

73.1.2 Location

driver/memory/emif.c

73.1.3 Supported SoCs:

TI OMAP44xx TI OMAP54xx

73.1.4 Menuconfig option:

Device Drivers

Memory devices Texas Instruments EMIF driver

73.1.5 Description

This driver is for the EMIF module available in Texas Instruments SoCs. EMIF is an SDRAM controller that, based on its revision, supports one or more of DDR2, DDR3, and LPDDR2 SDRAM protocols. This driver takes care of only LPDDR2 memories presently. The functions of the driver includes re-configuring AC timing parameters and other settings during frequency, voltage and temperature changes

73.1.6 Platform Data (see include/linux/platform_data/emif_plat.h)

DDR device details and other board dependent and SoC dependent information can be passed through platform data (struct emif_platform_data)

- DDR device details: 'struct ddr_device_info'
- Device AC timings: 'struct lpddr2_timings' and 'struct lpddr2_min_tck'
- Custom configurations: customizable policy options through 'struct emif_custom_configs'
- IP revision
- PHY type

73.1.7 Interface to the external world

EMIF driver registers notifiers for voltage and frequency changes affecting EMIF and takes appropriate actions when these are invoked.

- freq_pre_notify_handling()
- freq_post_notify_handling()
- volt_notify_handling()

73.1.8 Debugfs

The driver creates two debugfs entries per device.

- regcache_dump : dump of register values calculated and saved for all frequencies used so far.
- mr4 : last polled value of MR4 register in the LPDDR2 device. MR4 indicates the current temperature level of the device.

73.2 GPMC (General Purpose Memory Controller)

GPMC is an unified memory controller dedicated to interfacing external memory devices like

- Asynchronous SRAM like memories and application specific integrated circuit devices.
- Asynchronous, synchronous, and page mode burst NOR flash devices NAND flash
- Pseudo-SRAM devices

GPMC is found on Texas Instruments SoC's (OMAP based) IP details: http://www.ti.com/lit/pdf/spruh73 section 7.1

73.2.1 GPMC generic timing calculation:

GPMC has certain timings that has to be programmed for proper functioning of the peripheral, while peripheral has another set of timings. To have peripheral work with gpmc, peripheral timings has to be translated to the form gpmc can understand. The way it has to be translated depends on the connected peripheral. Also there is a dependency for certain gpmc timings on gpmc clock frequency. Hence a generic timing routine was developed to achieve above requirements.

Generic routine provides a generic method to calculate gpmc timings from gpmc peripheral timings. struct gpmc_device_timings fields has to be updated with timings from the datasheet of the peripheral that is connected to gpmc. A few of the peripheral timings can be fed either in time or in cycles, provision to handle this scenario has been provided (refer struct gpmc_device_timings definition). It may so happen that timing as specified by peripheral datasheet is not present in timing structure, in this scenario, try to correlate peripheral timing to the one available. If that doesn't work, try to add a new field as required by peripheral, educate generic timing routine to handle it, make sure that it does not break any of the existing. Then there may be cases where peripheral datasheet doesn't mention certain fields of struct gpmc_device_timings, zero those entries.

Generic timing routine has been verified to work properly on multiple onenand's and tusb6010 peripherals.

A word of caution: generic timing routine has been developed based on understanding of gpmc timings, peripheral timings, available custom timing routines, a kind of reverse engineering without most of the datasheets & hardware (to be exact none of those supported in mainline having custom timing routine) and by simulation.

gpmc timing dependency on peripheral timings:

[<gpmc_timing>: <peripheral timing1>, <peripheral timing2> ⋯]

1. common

cs_on: t_ceasu

adv_on: t_avdasu, t_ceavd

2. sync common

sync_clk: clk

page_burst_access: t_bacc

clk_activation: t_ces, t_avds

3. read async muxed

adv_rd_off: t_avdp_r

oe_on: t_oeasu, t_aavdh

access: t_iaa, t_oe, t_ce, t_aa

rd_cycle: t_rd_cycle, t_cez_r, t_oez

4. read async non-muxed

adv_rd_off: t_avdp_r

oe_on: t_oeasu

access: t_iaa, t_oe, t_ce, t_aa

rd_cycle: t_rd_cycle, t_cez_r, t_oez

5. read sync muxed

adv_rd_off: t_avdp_r, t_avdh

oe_on: t_oeasu, t_ach, cyc_aavdh_oe

access: t_iaa, cyc_iaa, cyc_oe

rd_cycle: t_cez_r, t_oez, t_ce_rdyz

6. read sync non-muxed

adv_rd_off: t_avdp_r

oe_on: t_oeasu

access: t_iaa, cyc_iaa, cyc_oe

rd_cycle: t_cez_r, t_oez, t_ce_rdyz

7. write async muxed

adv_wr_off: t_avdp_w

we_on, wr_data_mux_bus: t_weasu, t_aavdh, cyc_aavhd_we

we_off: t_wpl

cs_wr_off: t_wph

wr_cycle: t_cez_w, t_wr_cycle

8. write async non-muxed

adv_wr_off: t_avdp_w

we_on, wr_data_mux_bus: t_weasu

we_off: t_wpl

cs_wr_off: t_wph

wr_cycle: t_cez_w, t_wr_cycle

9. write sync muxed

adv_wr_off: t_avdp_w, t_avdh

we_on, wr_data_mux_bus: t_weasu, t_rdyo, t_aavdh, cyc_aavhd_we

we_off: t_wpl, cyc_wpl

cs_wr_off: t_wph

wr_cycle: t_cez_w, t_ce_rdyz

10. write sync non-muxed

adv_wr_off: t_avdp_w

we_on, wr_data_mux_bus: t_weasu, t_rdyo

we_off: t_wpl, cyc_wpl

cs_wr_off: t_wph

wr_cycle: t_cez_w, t_ce_rdyz

Note: Many of gpmc timings are dependent on other gpmc timings (a few gpmc timings purely dependent on other gpmc timings, a reason that some of the gpmc timings are missing above), and it will result in indirect dependency of peripheral timings to gpmc timings other than mentioned above, refer timing routine for more details. To know what these peripheral timings correspond to, please see explanations in struct gpmc_device_timings definition. And for gpmc timings refer IP details (link above).

CHAPTER

SEVENTYFOUR

MEN CHAMELEON BUS

74.1 Introduction

This document describes the architecture and implementation of the MEN Chameleon Bus (called MCB throughout this document).

74.1.1 Scope of this Document

This document is intended to be a short overview of the current implementation and does by no means describe the complete possibilities of MCB based devices.

74.1.2 Limitations of the current implementation

The current implementation is limited to PCI and PCIe based carrier devices that only use a single memory resource and share the PCI legacy IRQ. Not implemented are:

- Multi-resource MCB devices like the VME Controller or M-Module carrier.
- MCB devices that need another MCB device, like SRAM for a DMA Controller' s buffer descriptors or a video controller' s video memory.
- A per-carrier IRQ domain for carrier devices that have one (or more) IRQs per MCB device like PCIe based carriers with MSI or MSI-X support.

74.2 Architecture

MCB is divided into 3 functional blocks:

- The MEN Chameleon Bus itself,
- drivers for MCB Carrier Devices and
- the parser for the Chameleon table.

74.2.1 MEN Chameleon Bus

The MEN Chameleon Bus is an artificial bus system that attaches to a so called Chameleon FPGA device found on some hardware produced my MEN Mikro Elektronik GmbH. These devices are multi-function devices implemented in a single FPGA and usually attached via some sort of PCI or PCIe link. Each FPGA contains a header section describing the content of the FPGA. The header lists the device id, PCI BAR, offset from the beginning of the PCI BAR, size in the FPGA, interrupt number and some other properties currently not handled by the MCB implementation.

74.2.2 Carrier Devices

A carrier device is just an abstraction for the real world physical bus the Chameleon FPGA is attached to. Some IP Core drivers may need to interact with properties of the carrier device (like querying the IRQ number of a PCI device). To provide abstraction from the real hardware bus, an MCB carrier device provides callback methods to translate the driver's MCB function calls to hardware related function calls. For example a carrier device may implement the get_irq() method which can be translated into a hardware bus query for the IRQ number the device should use.

74.2.3 Parser

The parser reads the first 512 bytes of a Chameleon device and parses the Chameleon table. Currently the parser only supports the Chameleon v2 variant of the Chameleon table but can easily be adopted to support an older or possible future variant. While parsing the table's entries new MCB devices are allocated and their resources are assigned according to the resource assignment in the Chameleon table. After resource assignment is finished, the MCB devices are registered at the MCB and thus at the driver core of the Linux kernel.

74.3 Resource handling

The current implementation assigns exactly one memory and one IRQ resource per MCB device. But this is likely going to change in the future.

74.3.1 Memory Resources

Each MCB device has exactly one memory resource, which can be requested from the MCB bus. This memory resource is the physical address of the MCB device inside the carrier and is intended to be passed to `ioremap()` and friends. It is already requested from the kernel by calling request_mem_region().

74.3.2 IRQs

Each MCB device has exactly one IRQ resource, which can be requested from the MCB bus. If a carrier device driver implements the ->get_irq() callback method, the IRQ number assigned by the carrier device will be returned, otherwise the IRQ number inside the Chameleon table will be returned. This number is suitable to be passed to request_irq().

74.4 Writing an MCB driver

74.4.1 The driver structure

Each MCB driver has a structure to identify the device driver as well as device ids which identify the IP Core inside the FPGA. The driver structure also contains callback methods which get executed on driver probe and removal from the system:

```
static const struct mcb_device_id foo_ids[] = {
        { .device = 0x123 },
        { }
};
MODULE_DEVICE_TABLE(mcb, foo_ids);

static struct mcb_driver foo_driver = {
driver = {
        .name = "foo-bar",
        .owner = THIS_MODULE,
},
        .probe = foo_probe,
        .remove = foo_remove,
        .id_table = foo_ids,
};
```

74.4.2 Probing and attaching

When a driver is loaded and the MCB devices it services are found, the MCB core will call the driver's probe callback method. When the driver is removed from the system, the MCB core will call the driver's remove callback method:

```
static init foo_probe(struct mcb_device *mdev, const struct mcb_device_id␣
↪*id);
static void foo_remove(struct mcb_device *mdev);
```

74.4.3 Initializing the driver

When the kernel is booted or your foo driver module is inserted, you have to perform driver initialization. Usually it is enough to register your driver module at the MCB core:

```
static int __init foo_init(void)
{
        return mcb_register_driver(&foo_driver);
}
module_init(foo_init);

static void __exit foo_exit(void)
{
        mcb_unregister_driver(&foo_driver);
}
module_exit(foo_exit);
```

The module_mcb_driver() macro can be used to reduce the above code:

```
module_mcb_driver(foo_driver);
```

CHAPTER

SEVENTYFIVE

NTB DRIVERS

NTB (Non-Transparent Bridge) is a type of PCI-Express bridge chip that connects the separate memory systems of two or more computers to the same PCI-Express fabric. Existing NTB hardware supports a common feature set: doorbell registers and memory translation windows, as well as non common features like scratchpad and message registers. Scratchpad registers are read-and-writable registers that are accessible from either side of the device, so that peers can exchange a small amount of information at a fixed address. Message registers can be utilized for the same purpose. Additionally they are provided with with special status bits to make sure the information isn't rewritten by another peer. Doorbell registers provide a way for peers to send interrupt events. Memory windows allow translated read and write access to the peer memory.

75.1 NTB Core Driver (ntb)

The NTB core driver defines an api wrapping the common feature set, and allows clients interested in NTB features to discover NTB the devices supported by hardware drivers. The term "client" is used here to mean an upper layer component making use of the NTB api. The term "driver," or "hardware driver," is used here to mean a driver for a specific vendor and model of NTB hardware.

75.2 NTB Client Drivers

NTB client drivers should register with the NTB core driver. After registering, the client probe and remove functions will be called appropriately as ntb hardware, or hardware drivers, are inserted and removed. The registration uses the Linux Device framework, so it should feel familiar to anyone who has written a pci driver.

75.2.1 NTB Typical client driver implementation

Primary purpose of NTB is to share some peace of memory between at least two systems. So the NTB device features like Scratchpad/Message registers are mainly used to perform the proper memory window initialization. Typically there are two types of memory window interfaces supported by the NTB API: inbound translation configured on the local ntb port and outbound translation configured by the peer, on the peer ntb port. The first type is depicted on the next figure:

```
Inbound translation:

Memory:              Local NTB Port:      Peer NTB Port:      Peer MMIO:

 _____________
| dma-mapped |-ntb_mw_set_trans(addr)  |
| memory     |         _v_____________   |   _______________
| (addr)     |<======| MW xlat addr |<====| MW base addr |<== memory-
↪mapped IO
|------------|       |--------------|  |  |--------------|
```

So typical scenario of the first type memory window initialization looks: 1) allocate a memory region, 2) put translated address to NTB config, 3) somehow notify a peer device of performed initialization, 4) peer device maps corresponding outbound memory window so to have access to the shared memory region.

The second type of interface, that implies the shared windows being initialized by a peer device, is depicted on the figure:

```
Outbound translation:

Memory:        Local NTB Port:    Peer NTB Port:      Peer MMIO:

 _____________
| dma-mapped |                |   | MW base addr |<== memory-mapped IO
| memory     |                |   |--------------|
| (addr)     |<===============| MW xlat addr |<-ntb_peer_mw_set_
↪trans(addr)
|------------|                |   |--------------|
```

Typical scenario of the second type interface initialization would be: 1) allocate a memory region, 2) somehow deliver a translated address to a peer device, 3) peer puts the translated address to NTB config, 4) peer device maps outbound memory window so to have access to the shared memory region.

As one can see the described scenarios can be combined in one portable algorithm.

Local device:

1) Allocate memory for a shared window

2) Initialize memory window by translated address of the allocated region (it may fail if local memory window initialization is unsupported)

3) Send the translated address and memory window index to a peer device

Peer device:

1) Initialize memory window with retrieved address of the allocated by another device memory region (it may fail if peer memory window initialization is unsupported)

2) Map outbound memory window

In accordance with this scenario, the NTB Memory Window API can be used as follows:

Local device:

1) ntb_mw_count(pidx) - retrieve number of memory ranges, which can be allocated for memory windows between local device and peer device of port with specified index.

2) ntb_get_align(pidx, midx) - retrieve parameters restricting the shared memory region alignment and size. Then memory can be properly allocated.

3) Allocate physically contiguous memory region in compliance with restrictions retrieved in 2).

4) ntb_mw_set_trans(pidx, midx) - try to set translation address of the memory window with specified index for the defined peer device (it may fail if local translated address setting is not supported)

5) Send translated base address (usually together with memory window number) to the peer device using, for instance, scratchpad or message registers.

Peer device:

1) ntb_peer_mw_set_trans(pidx, midx) - try to set received from other device (related to pidx) translated address for specified memory window. It may fail if retrieved address, for instance, exceeds maximum possible address or isn't properly aligned.

2) ntb_peer_mw_get_addr(widx) - retrieve MMIO address to map the memory window so to have an access to the shared memory.

Also it is worth to note, that method ntb_mw_count(pidx) should return the same value as ntb_peer_mw_count() on the peer with port index - pidx.

75.2.2 NTB Transport Client (ntb_transport) and NTB Netdev (ntb_netdev)

The primary client for NTB is the Transport client, used in tandem with NTB Netdev. These drivers function together to create a logical link to the peer, across the ntb, to exchange packets of network data. The Transport client establishes a logical link to the peer, and creates queue pairs to exchange messages and data. The NTB Netdev then creates an ethernet device using a Transport queue pair. Network data is copied between socket buffers and the Transport queue pair buffer. The Transport client may be used for other things besides Netdev, however no other applications have yet been written.

75.2.3 NTB Ping Pong Test Client (ntb_pingpong)

The Ping Pong test client serves as a demonstration to exercise the doorbell and scratchpad registers of NTB hardware, and as an example simple NTB client. Ping Pong enables the link when started, waits for the NTB link to come up, and then proceeds to read and write the doorbell scratchpad registers of the NTB. The peers interrupt each other using a bit mask of doorbell bits, which is shifted by one in each round, to test the behavior of multiple doorbell bits and interrupt vectors. The Ping Pong driver also reads the first local scratchpad, and writes the value plus one to the first peer scratchpad, each round before writing the peer doorbell register.

Module Parameters:

- **unsafe - Some hardware has known issues with scratchpad and doorbell** registers. By default, Ping Pong will not attempt to exercise such hardware. You may override this behavior at your own risk by setting unsafe=1.
- **delay_ms - Specify the delay between receiving a doorbell** interrupt event and setting the peer doorbell register for the next round.
- **init_db - Specify the doorbell bits to start new series of rounds. A new** series begins once all the doorbell bits have been shifted out of range.
- **dyndbg - It is suggested to specify dyndbg=+p when loading this module, and** then to observe debugging output on the console.

75.2.4 NTB Tool Test Client (ntb_tool)

The Tool test client serves for debugging, primarily, ntb hardware and drivers. The Tool provides access through debugfs for reading, setting, and clearing the NTB doorbell, and reading and writing scratchpads.

The Tool does not currently have any module parameters.

Debugfs Files:

- **debugfs/ntb_tool/hw/** A directory in debugfs will be created for each NTB device probed by the tool. This directory is shortened to hw below.

- **hw/db** This file is used to read, set, and clear the local doorbell. Not all operations may be supported by all hardware. To read the doorbell, read the file. To set the doorbell, write s followed by the bits to set (eg: echo 's 0x0101' > db). To clear the doorbell, write c followed by the bits to clear.
- **hw/mask** This file is used to read, set, and clear the local doorbell mask. See db for details.
- **hw/peer_db** This file is used to read, set, and clear the peer doorbell. See db for details.
- **hw/peer_mask** This file is used to read, set, and clear the peer doorbell mask. See db for details.
- **hw/spad** This file is used to read and write local scratchpads. To read the values of all scratchpads, read the file. To write values, write a series of pairs of scratchpad number and value (eg: echo '4 0x123 7 0xabc' > spad # to set scratchpads 4 and 7 to 0x123 and 0xabc, respectively).
- **hw/peer_spad** This file is used to read and write peer scratchpads. See spad for details.

75.2.5 NTB MSI Test Client (ntb_msi_test)

The MSI test client serves to test and debug the MSI library which allows for passing MSI interrupts across NTB memory windows. The test client is interacted with through the debugfs filesystem:

- **debugfs/ntb_tool/hw/** A directory in debugfs will be created for each NTB device probed by the tool. This directory is shortened to hw below.
- **hw/port** This file describes the local port number
- **hw/irq*_occurrences** One occurrences file exists for each interrupt and, when read, returns the number of times the interrupt has been triggered.
- **hw/peer*/port** This file describes the port number for each peer
- **hw/peer*/count** This file describes the number of interrupts that can be triggered on each peer
- **hw/peer*/trigger** Writing an interrupt number (any number less than the value specified in count) will trigger the interrupt on the specified peer. That peer's interrupt's occurrence file should be incremented.

75.3 NTB Hardware Drivers

NTB hardware drivers should register devices with the NTB core driver. After registering, clients probe and remove functions will be called.

75.3.1 NTB Intel Hardware Driver (ntb_hw_intel)

The Intel hardware driver supports NTB on Xeon and Atom CPUs.

Module Parameters:

- **b2b_mw_idx** If the peer ntb is to be accessed via a memory window, then use this memory window to access the peer ntb. A value of zero or positive starts from the first mw idx, and a negative value starts from the last mw idx. Both sides MUST set the same value here! The default value is -1.
- **b2b_mw_share** If the peer ntb is to be accessed via a memory window, and if the memory window is large enough, still allow the client to use the second half of the memory window for address translation to the peer.
- **xeon_b2b_usd_bar2_addr64** If using B2B topology on Xeon hardware, use this 64 bit address on the bus between the NTB devices for the window at BAR2, on the upstream side of the link.
- xeon_b2b_usd_bar4_addr64 - See xeon_b2b_bar2_addr64.
- xeon_b2b_usd_bar4_addr32 - See xeon_b2b_bar2_addr64.
- xeon_b2b_usd_bar5_addr32 - See xeon_b2b_bar2_addr64.
- xeon_b2b_dsd_bar2_addr64 - See xeon_b2b_bar2_addr64.
- xeon_b2b_dsd_bar4_addr64 - See xeon_b2b_bar2_addr64.
- xeon_b2b_dsd_bar4_addr32 - See xeon_b2b_bar2_addr64.
- xeon_b2b_dsd_bar5_addr32 - See xeon_b2b_bar2_addr64.

CHAPTER

SEVENTYSIX

NVMEM SUBSYSTEM

Srinivas Kandagatla <srinivas.kandagatla@linaro.org>

This document explains the NVMEM Framework along with the APIs provided, and how to use it.

76.1 1. Introduction

NVMEM is the abbreviation for Non Volatile Memory layer. It is used to retrieve configuration of SOC or Device specific data from non volatile memories like eeprom, efuses and so on.

Before this framework existed, NVMEM drivers like eeprom were stored in drivers/misc, where they all had to duplicate pretty much the same code to register a sysfs file, allow in-kernel users to access the content of the devices they were driving, etc.

This was also a problem as far as other in-kernel users were involved, since the solutions used were pretty much different from one driver to another, there was a rather big abstraction leak.

This framework aims at solve these problems. It also introduces DT representation for consumer devices to go get the data they require (MAC Addresses, SoC/Revision ID, part numbers, and so on) from the NVMEMs. This framework is based on regmap, so that most of the abstraction available in regmap can be reused, across multiple types of buses.

76.1.1 NVMEM Providers

NVMEM provider refers to an entity that implements methods to initialize, read and write the non-volatile memory.

76.2 2. Registering/Unregistering the NVMEM provider

A NVMEM provider can register with NVMEM core by supplying relevant nvmem configuration to nvmem_register(), on success core would return a valid nvmem_device pointer.

nvmem_unregister(nvmem) is used to unregister a previously registered provider.

For example, a simple qfprom case:

```
static struct nvmem_config econfig = {
      .name = "qfprom",
      .owner = THIS_MODULE,
};

static int qfprom_probe(struct platform_device *pdev)
{
      ...
      econfig.dev = &pdev->dev;
      nvmem = nvmem_register(&econfig);
      ...
}
```

It is mandatory that the NVMEM provider has a regmap associated with its struct device. Failure to do would return error code from nvmem_register().

Users of board files can define and register nvmem cells using the nvmem_cell_table struct:

```
static struct nvmem_cell_info foo_nvmem_cells[] = {
      {
              .name           = "macaddr",
              .offset         = 0x7f00,
              .bytes          = ETH_ALEN,
      }
};

static struct nvmem_cell_table foo_nvmem_cell_table = {
      .nvmem_name             = "i2c-eeprom",
      .cells                  = foo_nvmem_cells,
      .ncells                 = ARRAY_SIZE(foo_nvmem_cells),
};

nvmem_add_cell_table(&foo_nvmem_cell_table);
```

Additionally it is possible to create nvmem cell lookup entries and register them with the nvmem framework from machine code as shown in the example below:

```
static struct nvmem_cell_lookup foo_nvmem_lookup = {
      .nvmem_name             = "i2c-eeprom",
      .cell_name              = "macaddr",
      .dev_id                 = "foo_mac.0",
      .con_id                 = "mac-address",
};

nvmem_add_cell_lookups(&foo_nvmem_lookup, 1);
```

76.2.1 NVMEM Consumers

NVMEM consumers are the entities which make use of the NVMEM provider to read from and to NVMEM.

76.3 3. NVMEM cell based consumer APIs

NVMEM cells are the data entries/fields in the NVMEM. The NVMEM framework provides 3 APIs to read/write NVMEM cells:

```
struct nvmem_cell *nvmem_cell_get(struct device *dev, const char *name);
struct nvmem_cell *devm_nvmem_cell_get(struct device *dev, const char␣
↪*name);

void nvmem_cell_put(struct nvmem_cell *cell);
void devm_nvmem_cell_put(struct device *dev, struct nvmem_cell *cell);

void *nvmem_cell_read(struct nvmem_cell *cell, ssize_t *len);
int nvmem_cell_write(struct nvmem_cell *cell, void *buf, ssize_t len);
```

*nvmem_cell_get() apis will get a reference to nvmem cell for a given id, and nvmem_cell_read/write() can then read or write to the cell. Once the usage of the cell is finished the consumer should call *nvmem_cell_put() to free all the allocation memory for the cell.

76.4 4. Direct NVMEM device based consumer APIs

In some instances it is necessary to directly read/write the NVMEM. To facilitate such consumers NVMEM framework provides below apis:

```
struct nvmem_device *nvmem_device_get(struct device *dev, const char␣
↪*name);
struct nvmem_device *devm_nvmem_device_get(struct device *dev,
                                           const char *name);
struct nvmem_device *nvmem_device_find(void *data,
                       int (*match)(struct device *dev, const void *data));
void nvmem_device_put(struct nvmem_device *nvmem);
int nvmem_device_read(struct nvmem_device *nvmem, unsigned int offset,
                      size_t bytes, void *buf);
int nvmem_device_write(struct nvmem_device *nvmem, unsigned int offset,
                       size_t bytes, void *buf);
int nvmem_device_cell_read(struct nvmem_device *nvmem,
                           struct nvmem_cell_info *info, void *buf);
int nvmem_device_cell_write(struct nvmem_device *nvmem,
                            struct nvmem_cell_info *info, void *buf);
```

Before the consumers can read/write NVMEM directly, it should get hold of nvmem_controller from one of the *nvmem_device_get() api.

The difference between these apis and cell based apis is that these apis always take nvmem_device as parameter.

76.5 5. Releasing a reference to the NVMEM

When a consumer no longer needs the NVMEM, it has to release the reference to the NVMEM it has obtained using the APIs mentioned in the above section. The NVMEM framework provides 2 APIs to release a reference to the NVMEM:

```
void nvmem_cell_put(struct nvmem_cell *cell);
void devm_nvmem_cell_put(struct device *dev, struct nvmem_cell *cell);
void nvmem_device_put(struct nvmem_device *nvmem);
void devm_nvmem_device_put(struct device *dev, struct nvmem_device *nvmem);
```

Both these APIs are used to release a reference to the NVMEM and devm_nvmem_cell_put and devm_nvmem_device_put destroys the devres associated with this NVMEM.

76.5.1 Userspace

76.6 6. Userspace binary interface

Userspace can read/write the raw NVMEM file located at:

```
/sys/bus/nvmem/devices/*/nvmem
```

ex:

```
hexdump /sys/bus/nvmem/devices/qfprom0/nvmem

0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
00000a0 db10 2240 0000 e000 0c00 0c00 0000 0c00
0000000 0000 0000 0000 0000 0000 0000 0000 0000
...
*
0001000
```

76.7 7. DeviceTree Binding

See Documentation/devicetree/bindings/nvmem/nvmem.txt

CHAPTER

SEVENTYSEVEN

PARPORT INTERFACE DOCUMENTATION

Time-stamp <2000-02-24 13:30:20 twaugh>

Described here are the following functions:

Global functions:: parport_register_driver parport_unregister_driver parport_enumerate parport_register_device parport_unregister_device parport_claim parport_claim_or_block parport_release parport_yield parport_yield_blocking parport_wait_peripheral parport_poll_peripheral parport_wait_event parport_negotiate parport_read parport_write parport_open parport_close parport_device_id parport_device_coords parport_find_class parport_find_device parport_set_timeout

Port functions (can be overridden by low-level drivers):

SPP:: port->ops->read_data port->ops->write_data port->ops->read_status port->ops->read_control port->ops->write_control port->ops->frob_control port->ops->enable_irq port->ops->disable_irq port->ops->data_forward port->ops->data_reverse

EPP:: port->ops->epp_write_data port->ops->epp_read_data port->ops->epp_write_addr port->ops->epp_read_addr

ECP:: port->ops->ecp_write_data port->ops->ecp_read_data port->ops->ecp_write_addr

Other:: port->ops->nibble_read_data port->ops->byte_read_data port->ops->compat_write_data

The parport subsystem comprises `parport` (the core port-sharing code), and a variety of low-level drivers that actually do the port accesses. Each low-level driver handles a particular style of port (PC, Amiga, and so on).

The parport interface to the device driver author can be broken down into global functions and port functions.

The global functions are mostly for communicating between the device driver and the parport subsystem: acquiring a list of available ports, claiming a port for exclusive use, and so on. They also include `generic` functions for doing standard things that will work on any IEEE 1284-capable architecture.

The port functions are provided by the low-level drivers, although the core parport module provides generic `defaults` for some routines. The port functions can be split into three groups: SPP, EPP, and ECP.

SPP (Standard Parallel Port) functions modify so-called SPP registers: data, status, and control. The hardware may not actually have registers exactly like that, but the PC does and this interface is modelled after common PC implementations. Other low-level drivers may be able to emulate most of the functionality.

EPP (Enhanced Parallel Port) functions are provided for reading and writing in IEEE 1284 EPP mode, and ECP (Extended Capabilities Port) functions are used for IEEE 1284 ECP mode. (What about BECP? Does anyone care?)

Hardware assistance for EPP and/or ECP transfers may or may not be available, and if it is available it may or may not be used. If hardware is not used, the transfer will be software-driven. In order to cope with peripherals that only tenuously support IEEE 1284, a low-level driver specific function is provided, for altering 'fudge factors'.

77.1 Global functions

77.1.1 parport_register_driver - register a device driver with parport

SYNOPSIS

```
#include <linux/parport.h>

struct parport_driver {
        const char *name;
        void (*attach) (struct parport *);
        void (*detach) (struct parport *);
        struct parport_driver *next;
};
int parport_register_driver (struct parport_driver *driver);
```

DESCRIPTION

In order to be notified about parallel ports when they are detected, parport_register_driver should be called. Your driver will immediately be notified of all ports that have already been detected, and of each new port as low-level drivers are loaded.

A `struct parport_driver` contains the textual name of your driver, a pointer to a function to handle new ports, and a pointer to a function to handle ports going away due to a low-level driver unloading. Ports will only be detached if they are not being used (i.e. there are no devices registered on them).

The visible parts of the `struct parport *` argument given to attach/detach are:

```
struct parport
{
        struct parport *next; /* next parport in list */
        const char *name;     /* port's name */
        unsigned int modes;   /* bitfield of hardware modes */
```

(continues on next page)

(continued from previous page)

```
        struct parport_device_info probe_info;
                        /* IEEE1284 info */
        int number;           /* parport index */
        struct parport_operations *ops;
        ...
};
```

There are other members of the structure, but they should not be touched.

The `modes` member summarises the capabilities of the underlying hardware. It consists of flags which may be bitwise-ored together:

PARPORT_MODE_PCSPP	IBM PC registers are available, i.e. functions that act on data, control and status registers are probably writing directly to the hardware.
PARPORT_MODE_TRISTATE	The data drivers may be turned off. This allows the data lines to be used for reverse (peripheral to host) transfers.
PARPORT_MODE_COMPAT	The hardware can assist with compatibility-mode (printer) transfers, i.e. compat_write_block.
PARPORT_MODE_EPP	The hardware can assist with EPP transfers.
PARPORT_MODE_ECP	The hardware can assist with ECP transfers.
PARPORT_MODE_DMA	The hardware can use DMA, so you might want to pass it DMA-able memory (i.e. memory allocated using the GFP_DMA flag with kmalloc) to the low-level driver in order to take advantage of it.

There may be other flags in `modes` as well.

The contents of `modes` is advisory only. For example, if the hardware is capable of DMA, and PARPORT_MODE_DMA is in `modes`, it doesn't necessarily mean that DMA will always be used when possible. Similarly, hardware that is capable of assisting ECP transfers won't necessarily be used.

RETURN VALUE

Zero on success, otherwise an error code.

ERRORS

None. (Can it fail? Why return int?)

EXAMPLE

```
static void lp_attach (struct parport *port)
{
        ...
        private = kmalloc (...);
        dev[count++] = parport_register_device (...);
        ...
}

static void lp_detach (struct parport *port)
{
        ...
}

static struct parport_driver lp_driver = {
        "lp",
        lp_attach,
        lp_detach,
        NULL /* always put NULL here */
};

int lp_init (void)
{
        ...
        if (parport_register_driver (&lp_driver)) {
                /* Failed; nothing we can do. */
                return -EIO;
        }
        ...
}
```

SEE ALSO

parport_unregister_driver, parport_register_device, parport_enumerate

77.1.2 parport_unregister_driver - tell parport to forget about this driver

SYNOPSIS

```
#include <linux/parport.h>

struct parport_driver {
        const char *name;
        void (*attach) (struct parport *);
        void (*detach) (struct parport *);
        struct parport_driver *next;
};
void parport_unregister_driver (struct parport_driver *driver);
```

DESCRIPTION

This tells parport not to notify the device driver of new ports or of ports going away. Registered devices belonging to that driver are NOT unregistered: parport_unregister_device must be used for each one.

EXAMPLE

```
void cleanup_module (void)
{
        ...
        /* Stop notifications. */
        parport_unregister_driver (&lp_driver);

        /* Unregister devices. */
        for (i = 0; i < NUM_DEVS; i++)
                parport_unregister_device (dev[i]);
        ...
}
```

SEE ALSO

parport_register_driver, parport_enumerate

77.1.3 parport_enumerate - retrieve a list of parallel ports (DEPRECATED)

SYNOPSIS

```
#include <linux/parport.h>

struct parport *parport_enumerate (void);
```

DESCRIPTION

Retrieve the first of a list of valid parallel ports for this machine. Successive parallel ports can be found using the `struct parport *next` element of the `struct parport *` that is returned. If `next` is NULL, there are no more parallel ports in the list. The number of ports in the list will not exceed PARPORT_MAX.

RETURN VALUE

A `struct parport *` describing a valid parallel port for the machine, or NULL if there are none.

ERRORS

This function can return NULL to indicate that there are no parallel ports to use.

EXAMPLE

```
int detect_device (void)
{
        struct parport *port;

        for (port = parport_enumerate ();
        port != NULL;
        port = port->next) {
                /* Try to detect a device on the port... */
                ...
        }
        }

        ...
}
```

NOTES

parport_enumerate is deprecated; parport_register_driver should be used instead.

SEE ALSO

parport_register_driver, parport_unregister_driver

77.1.4 parport_register_device - register to use a port

SYNOPSIS

```
#include <linux/parport.h>

typedef int (*preempt_func) (void *handle);
typedef void (*wakeup_func) (void *handle);
typedef int (*irq_func) (int irq, void *handle, struct pt_regs *);

struct pardevice *parport_register_device(struct parport *port,
                                          const char *name,
                                          preempt_func preempt,
                                          wakeup_func wakeup,
                                          irq_func irq,
```

(continues on next page)

(continued from previous page)

```
                                                int flags,
                                                void *handle);
```

DESCRIPTION

Use this function to register your device driver on a parallel port (`port`). Once you have done that, you will be able to use parport_claim and parport_release in order to use the port.

The (`name`) argument is the name of the device that appears in /proc filesystem. The string must be valid for the whole lifetime of the device (until parport_unregister_device is called).

This function will register three callbacks into your driver: `preempt`, `wakeup` and `irq`. Each of these may be NULL in order to indicate that you do not want a callback.

When the `preempt` function is called, it is because another driver wishes to use the parallel port. The `preempt` function should return non-zero if the parallel port cannot be released yet – if zero is returned, the port is lost to another driver and the port must be re-claimed before use.

The `wakeup` function is called once another driver has released the port and no other driver has yet claimed it. You can claim the parallel port from within the `wakeup` function (in which case the claim is guaranteed to succeed), or choose not to if you don't need it now.

If an interrupt occurs on the parallel port your driver has claimed, the `irq` function will be called. (Write something about shared interrupts here.)

The `handle` is a pointer to driver-specific data, and is passed to the callback functions.

`flags` may be a bitwise combination of the following flags:

Flag	Meaning
PARPORT_DEV_EXCL	The device cannot share the parallel port at all. Use this only when absolutely necessary.

The typedefs are not actually defined – they are only shown in order to make the function prototype more readable.

The visible parts of the returned `struct pardevice` are:

```
struct pardevice {
        struct parport *port;   /* Associated port */
        void *private;          /* Device driver's 'handle' */
        ...
};
```

RETURN VALUE

A `struct pardevice *`: a handle to the registered parallel port device that can be used for parport_claim, parport_release, etc.

ERRORS

A return value of NULL indicates that there was a problem registering a device on that port.

EXAMPLE

```
static int preempt (void *handle)
{
        if (busy_right_now)
                return 1;

        must_reclaim_port = 1;
        return 0;
}

static void wakeup (void *handle)
{
        struct toaster *private = handle;
        struct pardevice *dev = private->dev;
        if (!dev) return; /* avoid races */

        if (want_port)
                parport_claim (dev);
}

static int toaster_detect (struct toaster *private, struct parport *port)
{
        private->dev = parport_register_device (port, "toaster", preempt,
                                                wakeup, NULL, 0,
                                                private);
        if (!private->dev)
                /* Couldn't register with parport. */
                return -EIO;

        must_reclaim_port = 0;
        busy_right_now = 1;
        parport_claim_or_block (private->dev);
        ...
        /* Don't need the port while the toaster warms up. */
        busy_right_now = 0;
        ...
        busy_right_now = 1;
        if (must_reclaim_port) {
                parport_claim_or_block (private->dev);
                must_reclaim_port = 0;
        }
        ...
}
```

SEE ALSO

parport_unregister_device, parport_claim

77.1.5 parport_unregister_device - finish using a port

SYNPOPSIS

```
#include <linux/parport.h>

void parport_unregister_device (struct pardevice *dev);
```

DESCRIPTION

This function is the opposite of parport_register_device. After using parport_unregister_device, dev is no longer a valid device handle.

You should not unregister a device that is currently claimed, although if you do it will be released automatically.

EXAMPLE

```
...
kfree (dev->private); /* before we lose the pointer */
parport_unregister_device (dev);
...
```

SEE ALSO

parport_unregister_driver

77.1.6 parport_claim, parport_claim_or_block - claim the parallel port for a device

SYNOPSIS

```
#include <linux/parport.h>

int parport_claim (struct pardevice *dev);
int parport_claim_or_block (struct pardevice *dev);
```

DESCRIPTION

These functions attempt to gain control of the parallel port on which `dev` is registered. `parport_claim` does not block, but `parport_claim_or_block` may do. (Put something here about blocking interruptibly or non-interruptibly.)

You should not try to claim a port that you have already claimed.

RETURN VALUE

A return value of zero indicates that the port was successfully claimed, and the caller now has possession of the parallel port.

If `parport_claim_or_block` blocks before returning successfully, the return value is positive.

ERRORS

-EAGAIN	The port is unavailable at the moment, but another attempt to claim it may succeed.

SEE ALSO

parport_release

77.1.7 parport_release - release the parallel port

SYNOPSIS

```
#include <linux/parport.h>

void parport_release (struct pardevice *dev);
```

DESCRIPTION

Once a parallel port device has been claimed, it can be released using `parport_release`. It cannot fail, but you should not release a device that you do not have possession of.

EXAMPLE

```
static size_t write (struct pardevice *dev, const void *buf,
                 size_t len)
{
        ...
        written = dev->port->ops->write_ecp_data (dev->port, buf,
                                                  len);
        parport_release (dev);
        ...
}
```

SEE ALSO

change_mode, parport_claim, parport_claim_or_block, parport_yield

77.1.8 parport_yield, parport_yield_blocking - temporarily release a parallel port

SYNOPSIS

```
#include <linux/parport.h>

int parport_yield (struct pardevice *dev)
int parport_yield_blocking (struct pardevice *dev);
```

DESCRIPTION

When a driver has control of a parallel port, it may allow another driver to temporarily `borrow` it. `parport_yield` does not block; `parport_yield_blocking` may do.

RETURN VALUE

A return value of zero indicates that the caller still owns the port and the call did not block.

A positive return value from `parport_yield_blocking` indicates that the caller still owns the port and the call blocked.

A return value of -EAGAIN indicates that the caller no longer owns the port, and it must be re-claimed before use.

ERRORS

-EAGAIN	Ownership of the parallel port was given away.

SEE ALSO

parport_release

77.1.9 parport_wait_peripheral - wait for status lines, up to 35ms

SYNOPSIS

```
#include <linux/parport.h>

int parport_wait_peripheral (struct parport *port,
                             unsigned char mask,
                             unsigned char val);
```

DESCRIPTION

Wait for the status lines in mask to match the values in val.

RETURN VALUE

-EINTR	a signal is pending
0	the status lines in mask have values in val
1	timed out while waiting (35ms elapsed)

SEE ALSO

parport_poll_peripheral

77.1.10 parport_poll_peripheral - wait for status lines, in usec

SYNOPSIS

```
#include <linux/parport.h>

int parport_poll_peripheral (struct parport *port,
                             unsigned char mask,
                             unsigned char val,
                             int usec);
```

DESCRIPTION

Wait for the status lines in mask to match the values in val.

RETURN VALUE

-EINTR	a signal is pending
0	the status lines in mask have values in val
1	timed out while waiting (usec microseconds have elapsed)

SEE ALSO

parport_wait_peripheral

77.1.11 parport_wait_event - wait for an event on a port

SYNOPSIS

```
#include <linux/parport.h>

int parport_wait_event (struct parport *port, signed long timeout)
```

DESCRIPTION

Wait for an event (e.g. interrupt) on a port. The timeout is in jiffies.

RETURN VALUE

0	success
<0	error (exit as soon as possible)
>0	timed out

77.1.12 parport_negotiate - perform IEEE 1284 negotiation

SYNOPSIS

```
#include <linux/parport.h>

int parport_negotiate (struct parport *, int mode);
```

DESCRIPTION

Perform IEEE 1284 negotiation.

RETURN VALUE

0	handshake OK; IEEE 1284 peripheral and mode available
-1	handshake failed; peripheral not compliant (or none present)
1	handshake OK; IEEE 1284 peripheral present but mode not available

SEE ALSO

parport_read, parport_write

77.1.13 parport_read - read data from device

SYNOPSIS

```
#include <linux/parport.h>

ssize_t parport_read (struct parport *, void *buf, size_t len);
```

DESCRIPTION

Read data from device in current IEEE 1284 transfer mode. This only works for modes that support reverse data transfer.

RETURN VALUE

If negative, an error code; otherwise the number of bytes transferred.

SEE ALSO

parport_write, parport_negotiate

77.1.14 parport_write - write data to device

SYNOPSIS

```
#include <linux/parport.h>

ssize_t parport_write (struct parport *, const void *buf, size_t len);
```

DESCRIPTION

Write data to device in current IEEE 1284 transfer mode. This only works for modes that support forward data transfer.

RETURN VALUE

If negative, an error code; otherwise the number of bytes transferred.

SEE ALSO

parport_read, parport_negotiate

77.1.15 parport_open - register device for particular device number

SYNOPSIS

```
#include <linux/parport.h>

struct pardevice *parport_open (int devnum, const char *name,
                                int (*pf) (void *),
                                void (*kf) (void *),
                                void (*irqf) (int, void *,
                                              struct pt_regs *),
                                int flags, void *handle);
```

DESCRIPTION

This is like parport_register_device but takes a device number instead of a pointer to a struct parport.

RETURN VALUE

See parport_register_device. If no device is associated with devnum, NULL is returned.

SEE ALSO

parport_register_device

77.1.16 parport_close - unregister device for particular device number

SYNOPSIS

```
#include <linux/parport.h>

void parport_close (struct pardevice *dev);
```

DESCRIPTION

This is the equivalent of parport_unregister_device for parport_open.

SEE ALSO

parport_unregister_device, parport_open

77.1.17 parport_device_id - obtain IEEE 1284 Device ID

SYNOPSIS

```
#include <linux/parport.h>

ssize_t parport_device_id (int devnum, char *buffer, size_t len);
```

DESCRIPTION

Obtains the IEEE 1284 Device ID associated with a given device.

RETURN VALUE

If negative, an error code; otherwise, the number of bytes of buffer that contain the device ID. The format of the device ID is as follows:

```
[length][ID]
```

The first two bytes indicate the inclusive length of the entire Device ID, and are in big-endian order. The ID is a sequence of pairs of the form:

```
key:value;
```

NOTES

Many devices have ill-formed IEEE 1284 Device IDs.

SEE ALSO

parport_find_class, parport_find_device

77.1.18 parport_device_coords - convert device number to device coordinates

SYNOPSIS

```
#include <linux/parport.h>

int parport_device_coords (int devnum, int *parport, int *mux,
                           int *daisy);
```

DESCRIPTION

Convert between device number (zero-based) and device coordinates (port, multiplexor, daisy chain address).

RETURN VALUE

Zero on success, in which case the coordinates are (`*parport`, `*mux`, `*daisy`).

SEE ALSO

parport_open, parport_device_id

77.1.19 parport_find_class - find a device by its class

SYNOPSIS

```
#include <linux/parport.h>

typedef enum {
        PARPORT_CLASS_LEGACY = 0,       /* Non-IEEE1284 device */
        PARPORT_CLASS_PRINTER,
        PARPORT_CLASS_MODEM,
        PARPORT_CLASS_NET,
        PARPORT_CLASS_HDC,              /* Hard disk controller */
        PARPORT_CLASS_PCMCIA,
        PARPORT_CLASS_MEDIA,            /* Multimedia device */
        PARPORT_CLASS_FDC,              /* Floppy disk controller */
        PARPORT_CLASS_PORTS,
```

(continues on next page)

(continued from previous page)

```
        PARPORT_CLASS_SCANNER,
        PARPORT_CLASS_DIGCAM,
        PARPORT_CLASS_OTHER,            /* Anything else */
        PARPORT_CLASS_UNSPEC,           /* No CLS field in ID */
        PARPORT_CLASS_SCSIADAPTER
} parport_device_class;

int parport_find_class (parport_device_class cls, int from);
```

DESCRIPTION

Find a device by class. The search starts from device number from+1.

RETURN VALUE

The device number of the next device in that class, or -1 if no such device exists.

NOTES

Example usage:

```
int devnum = -1;
while ((devnum = parport_find_class (PARPORT_CLASS_DIGCAM, devnum)) != -1)
↪{
        struct pardevice *dev = parport_open (devnum, ...);
        ...
}
```

SEE ALSO

parport_find_device, parport_open, parport_device_id

77.1.20 parport_find_device - find a device by its class

SYNOPSIS

```
#include <linux/parport.h>

int parport_find_device (const char *mfg, const char *mdl, int from);
```

DESCRIPTION

Find a device by vendor and model. The search starts from device number from+1.

RETURN VALUE

The device number of the next device matching the specifications, or -1 if no such device exists.

NOTES

Example usage:

```
int devnum = -1;
while ((devnum = parport_find_device ("IOMEGA", "ZIP+", devnum)) != -1) {
        struct pardevice *dev = parport_open (devnum, ...);
        ...
}
```

SEE ALSO

parport_find_class, parport_open, parport_device_id

77.1.21 parport_set_timeout - set the inactivity timeout

SYNOPSIS

```
#include <linux/parport.h>

long parport_set_timeout (struct pardevice *dev, long inactivity);
```

DESCRIPTION

Set the inactivity timeout, in jiffies, for a registered device. The previous timeout is returned.

RETURN VALUE

The previous timeout, in jiffies.

NOTES

Some of the port->ops functions for a parport may take time, owing to delays at the peripheral. After the peripheral has not responded for `inactivity` jiffies, a timeout will occur and the blocking function will return.

A timeout of 0 jiffies is a special case: the function must do as much as it can without blocking or leaving the hardware in an unknown state. If port operations are performed from within an interrupt handler, for instance, a timeout of 0 jiffies should be used.

Once set for a registered device, the timeout will remain at the set value until set again.

SEE ALSO

port->ops->xxx_read/write_yyy

77.2 PORT FUNCTIONS

The functions in the port->ops structure (struct parport_operations) are provided by the low-level driver responsible for that port.

77.2.1 port->ops->read_data - read the data register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        unsigned char (*read_data) (struct parport *port);
        ...
};
```

DESCRIPTION

If port->modes contains the PARPORT_MODE_TRISTATE flag and the PARPORT_CONTROL_DIRECTION bit in the control register is set, this returns the value on the data pins. If port->modes contains the PARPORT_MODE_TRISTATE flag and the PARPORT_CONTROL_DIRECTION bit is not set, the return value _may_ be the last value written to the data register. Otherwise the return value is undefined.

SEE ALSO

write_data, read_status, write_control

77.2.2 port->ops->write_data - write the data register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*write_data) (struct parport *port, unsigned char d);
        ...
};
```

DESCRIPTION

Writes to the data register. May have side-effects (a STROBE pulse, for instance).

SEE ALSO

read_data, read_status, write_control

77.2.3 port->ops->read_status - read the status register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        unsigned char (*read_status) (struct parport *port);
        ...
};
```

DESCRIPTION

Reads from the status register. This is a bitmask:

- PARPORT_STATUS_ERROR (printer fault, “nFault”)
- PARPORT_STATUS_SELECT (on-line, “Select”)
- PARPORT_STATUS_PAPEROUT (no paper, “PError”)
- PARPORT_STATUS_ACK (handshake, “nAck”)
- PARPORT_STATUS_BUSY (busy, “Busy”)

There may be other bits set.

SEE ALSO

read_data, write_data, write_control

77.2.4 port->ops->read_control - read the control register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        unsigned char (*read_control) (struct parport *port);
        ...
};
```

DESCRIPTION

Returns the last value written to the control register (either from write_control or frob_control). No port access is performed.

SEE ALSO

read_data, write_data, read_status, write_control

77.2.5 port->ops->write_control - write the control register

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*write_control) (struct parport *port, unsigned char s);
        ...
};
```

DESCRIPTION

Writes to the control register. This is a bitmask:

```
- PARPORT_CONTROL_STROBE (nStrobe)

- PARPORT_CONTROL_AUTOFD (nAutoFd)

- PARPORT_CONTROL_INIT (nInit)

- PARPORT_CONTROL_SELECT (nSelectIn)
```

SEE ALSO

read_data, write_data, read_status, frob_control

77.2.6 port->ops->frob_control - write control register bits

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        unsigned char (*frob_control) (struct parport *port,
                                       unsigned char mask,
                                       unsigned char val);
        ...
};
```

DESCRIPTION

This is equivalent to reading from the control register, masking out the bits in mask, exclusive-or'ing with the bits in val, and writing the result to the control register.

As some ports don't allow reads from the control port, a software copy of its contents is maintained, so frob_control is in fact only one port access.

SEE ALSO

read_data, write_data, read_status, write_control

77.2.7 port->ops->enable_irq - enable interrupt generation

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*enable_irq) (struct parport *port);
        ...
};
```

DESCRIPTION

The parallel port hardware is instructed to generate interrupts at appropriate moments, although those moments are architecture-specific. For the PC architecture, interrupts are commonly generated on the rising edge of nAck.

SEE ALSO

disable_irq

77.2.8 port->ops->disable_irq - disable interrupt generation

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*disable_irq) (struct parport *port);
        ...
};
```

DESCRIPTION

The parallel port hardware is instructed not to generate interrupts. The interrupt itself is not masked.

SEE ALSO

enable_irq

77.2.9 port->ops->data_forward - enable data drivers

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*data_forward) (struct parport *port);
        ...
};
```

DESCRIPTION

Enables the data line drivers, for 8-bit host-to-peripheral communications.

SEE ALSO

data_reverse

77.2.10 port->ops->data_reverse - tristate the buffer

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        void (*data_reverse) (struct parport *port);
        ...
};
```

DESCRIPTION

Places the data bus in a high impedance state, if port->modes has the PARPORT_MODE_TRISTATE bit set.

SEE ALSO

data_forward

77.2.11 port->ops->epp_write_data - write EPP data

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*epp_write_data) (struct parport *port, const void *buf,
                                  size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes data in EPP mode, and returns the number of bytes written.

The `flags` parameter may be one or more of the following, bitwise-or'ed together:

PARPORT_EPP_FAST	Use fast transfers. Some chips provide 16-bit and 32-bit registers. However, if a transfer times out, the return value may be unreliable.

SEE ALSO

epp_read_data, epp_write_addr, epp_read_addr

77.2.12 port->ops->epp_read_data - read EPP data

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*epp_read_data) (struct parport *port, void *buf,
                                 size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads data in EPP mode, and returns the number of bytes read.

The `flags` parameter may be one or more of the following, bitwise-or'ed together:

PARPORT_EPP_FAST	Use fast transfers. Some chips provide 16-bit and 32-bit registers. However, if a transfer times out, the return value may be unreliable.

SEE ALSO

epp_write_data, epp_write_addr, epp_read_addr

77.2.13 port->ops->epp_write_addr - write EPP address

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*epp_write_addr) (struct parport *port,
                                  const void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes EPP addresses (8 bits each), and returns the number written.

The `flags` parameter may be one or more of the following, bitwise-or'ed together:

PARPORT_EPP_FAST	Use fast transfers. Some chips provide 16-bit and 32-bit registers. However, if a transfer times out, the return value may be unreliable.

(Does PARPORT_EPP_FAST make sense for this function?)

SEE ALSO

epp_write_data, epp_read_data, epp_read_addr

77.2.14 port->ops->epp_read_addr - read EPP address

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*epp_read_addr) (struct parport *port, void *buf,
                                 size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads EPP addresses (8 bits each), and returns the number read.

The `flags` parameter may be one or more of the following, bitwise-or'ed together:

PARPORT_EPP_FAST	Use fast transfers. Some chips provide 16-bit and 32-bit registers. However, if a transfer times out, the return value may be unreliable.

(Does PARPORT_EPP_FAST make sense for this function?)

SEE ALSO

epp_write_data, epp_read_data, epp_write_addr

77.2.15 port->ops->ecp_write_data - write a block of ECP data

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*ecp_write_data) (struct parport *port,
                                  const void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes a block of ECP data. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes written.

SEE ALSO

ecp_read_data, ecp_write_addr

77.2.16 port->ops->ecp_read_data - read a block of ECP data

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*ecp_read_data) (struct parport *port,
                                 void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads a block of ECP data. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes read. NB. There may be more unread data in a FIFO. Is there a way of stunning the FIFO to prevent this?

SEE ALSO

ecp_write_block, ecp_write_addr

77.2.17 port->ops->ecp_write_addr - write a block of ECP addresses

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*ecp_write_addr) (struct parport *port,
                                  const void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes a block of ECP addresses. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes written.

NOTES

This may use a FIFO, and if so shall not return until the FIFO is empty.

SEE ALSO

ecp_read_data, ecp_write_data

77.2.18 port->ops->nibble_read_data - read a block of data in nibble mode

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*nibble_read_data) (struct parport *port,
                                    void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads a block of data in nibble mode. The `flags` parameter is ignored.

RETURN VALUE

The number of whole bytes read.

SEE ALSO

byte_read_data, compat_write_data

77.2.19 port->ops->byte_read_data - read a block of data in byte mode

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*byte_read_data) (struct parport *port,
                                  void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Reads a block of data in byte mode. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes read.

SEE ALSO

nibble_read_data, compat_write_data

77.2.20 port->ops->compat_write_data - write a block of data in compatibility mode

SYNOPSIS

```
#include <linux/parport.h>

struct parport_operations {
        ...
        size_t (*compat_write_data) (struct parport *port,
                                     const void *buf, size_t len, int flags);
        ...
};
```

DESCRIPTION

Writes a block of data in compatibility mode. The `flags` parameter is ignored.

RETURN VALUE

The number of bytes written.

SEE ALSO

nibble_read_data, byte_read_data

CHAPTER

SEVENTYEIGHT

PPS - PULSE PER SECOND

78.1 Overview

LinuxPPS provides a programming interface (API) to define in the system several PPS sources.

PPS means "pulse per second" and a PPS source is just a device which provides a high precision signal each second so that an application can use it to adjust system clock time.

A PPS source can be connected to a serial port (usually to the Data Carrier Detect pin) or to a parallel port (ACK-pin) or to a special CPU' s GPIOs (this is the common case in embedded systems) but in each case when a new pulse arrives the system must apply to it a timestamp and record it for userland.

Common use is the combination of the NTPD as userland program, with a GPS receiver as PPS source, to obtain a wallclock-time with sub-millisecond synchronisation to UTC.

78.2 RFC considerations

While implementing a PPS API as RFC 2783 defines and using an embedded CPU GPIO-Pin as physical link to the signal, I encountered a deeper problem:

> At startup it needs a file descriptor as argument for the function time_pps_create().

This implies that the source has a /dev/···entry. This assumption is OK for the serial and parallel port, where you can do something useful besides(!) the gathering of timestamps as it is the central task for a PPS API. But this assumption does not

work for a single purpose GPIO line. In this case even basic file-related functionality (like read() and write()) makes no sense at all and should not be a precondition for the use of a PPS API.

The problem can be simply solved if you consider that a PPS source is not always connected with a GPS data source.

So your programs should check if the GPS data source (the serial port for instance) is a PPS source too, and if not they should provide the possibility to open another device as PPS source.

In LinuxPPS the PPS sources are simply char devices usually mapped into files /dev/pps0, /dev/pps1, etc.

78.3 PPS with USB to serial devices

It is possible to grab the PPS from an USB to serial device. However, you should take into account the latencies and jitter introduced by the USB stack. Users have reported clock instability around +-1ms when synchronized with PPS through USB. With USB 2.0, jitter may decrease down to the order of 125 microseconds.

This may be suitable for time server synchronization with NTP because of its undersampling and algorithms.

If your device doesn't report PPS, you can check that the feature is supported by its driver. Most of the time, you only need to add a call to usb_serial_handle_dcd_change after checking the DCD status (see ch341 and pl2303 examples).

78.4 Coding example

To register a PPS source into the kernel you should define a struct pps_source_info as follows:

```
static struct pps_source_info pps_ktimer_info = {
        .name           = "ktimer",
        .path           = "",
        .mode           = PPS_CAPTUREASSERT | PPS_OFFSETASSERT |
                          PPS_ECHOASSERT |
                          PPS_CANWAIT | PPS_TSFMT_TSPEC,
        .echo           = pps_ktimer_echo,
        .owner          = THIS_MODULE,
};
```

and then calling the function pps_register_source() in your initialization routine as follows:

```
source = pps_register_source(&pps_ktimer_info,
                     PPS_CAPTUREASSERT | PPS_OFFSETASSERT);
```

The pps_register_source() prototype is:

```
int pps_register_source(struct pps_source_info *info, int default_params)
```

where "info" is a pointer to a structure that describes a particular PPS source, "default_params" tells the system what the initial default parameters for the device should be (it is obvious that these parameters must be a subset of ones defined in the struct pps_source_info which describe the capabilities of the driver).

Once you have registered a new PPS source into the system you can signal an assert event (for example in the interrupt handler routine) just using:

```
pps_event(source, &ts, PPS_CAPTUREASSERT, ptr)
```

where "ts" is the event's timestamp.

The same function may also run the defined echo function (pps_ktimer_echo(), passing to it the "ptr" pointer) if the user asked for that···etc..

Please see the file drivers/pps/clients/pps-ktimer.c for example code.

78.5 SYSFS support

If the SYSFS filesystem is enabled in the kernel it provides a new class:

```
$ ls /sys/class/pps/
pps0/  pps1/  pps2/
```

Every directory is the ID of a PPS sources defined in the system and inside you find several files:

```
$ ls -F /sys/class/pps/pps0/
assert     dev        mode       path       subsystem@
clear      echo       name       power/     uevent
```

Inside each "assert" and "clear" file you can find the timestamp and a sequence number:

```
$ cat /sys/class/pps/pps0/assert
1170026870.983207967#8
```

Where before the "#" is the timestamp in seconds; after it is the sequence number. Other files are:

- echo: reports if the PPS source has an echo function or not;
- mode: reports available PPS functioning modes;
- name: reports the PPS source's name;
- path: reports the PPS source's device path, that is the device the PPS source is connected to (if it exists).

78.6 Testing the PPS support

In order to test the PPS support even without specific hardware you can use the pps-ktimer driver (see the client subsection in the PPS configuration menu) and the userland tools available in your distribution's pps-tools package, http://linuxpps.org , or https://github.com/redlab-i/pps-tools.

Once you have enabled the compilation of pps-ktimer just modprobe it (if not statically compiled):

```
# modprobe pps-ktimer
```

and the run ppstest as follow:

```
$ ./ppstest /dev/pps1
trying PPS source "/dev/pps1"
found PPS source "/dev/pps1"
ok, found 1 source(s), now start fetching data...
source 0 - assert 1186592699.388832443, sequence: 364 - clear  0.000000000,
↪ sequence: 0
source 0 - assert 1186592700.388931295, sequence: 365 - clear  0.000000000,
↪ sequence: 0
source 0 - assert 1186592701.389032765, sequence: 366 - clear  0.000000000,
↪ sequence: 0
```

Please note that to compile userland programs, you need the file timepps.h. This is available in the pps-tools repository mentioned above.

78.7 Generators

Sometimes one needs to be able not only to catch PPS signals but to produce them also. For example, running a distributed simulation, which requires computers' clock to be synchronized very tightly. One way to do this is to invent some complicated hardware solutions but it may be neither necessary nor affordable. The cheap way is to load a PPS generator on one of the computers (master) and PPS clients on others (slaves), and use very simple cables to deliver signals using parallel ports, for example.

Parallel port cable pinout:

```
pin     name    master      slave
1       STROBE    *------     *
2       D0        *     |     *
3       D1        *     |     *
4       D2        *     |     *
5       D3        *     |     *
6       D4        *     |     *
7       D5        *     |     *
8       D6        *     |     *
9       D7        *     |     *
10      ACK       *     ------*
11      BUSY      *           *
12      PE        *           *
```

(continues on next page)

(continued from previous page)

```
13      SEL       *           *
14      AUTOFD    *           *
15      ERROR     *           *
16      INIT      *           *
17      SELIN     *           *
18-25   GND       *-----------*
```

Please note that parallel port interrupt occurs only on high->low transition, so it is used for PPS assert edge. PPS clear edge can be determined only using polling in the interrupt handler which actually can be done way more precisely because interrupt handling delays can be quite big and random. So current parport PPS generator implementation (pps_gen_parport module) is geared towards using the clear edge for time synchronization.

Clear edge polling is done with disabled interrupts so it' s better to select delay between assert and clear edge as small as possible to reduce system latencies. But if it is too small slave won' t be able to capture clear edge transition. The default of 30us should be good enough in most situations. The delay can be selected using 'delay' pps_gen_parport module parameter.

CHAPTER

SEVENTYNINE

PTP HARDWARE CLOCK INFRASTRUCTURE FOR LINUX

This patch set introduces support for IEEE 1588 PTP clocks in Linux. Together with the SO_TIMESTAMPING socket options, this presents a standardized method for developing PTP user space programs, synchronizing Linux with external clocks, and using the ancillary features of PTP hardware clocks.

A new class driver exports a kernel interface for specific clock drivers and a user space interface. The infrastructure supports a complete set of PTP hardware clock functionality.

- Basic clock operations - Set time - Get time - Shift the clock by a given offset atomically - Adjust clock frequency
- Ancillary clock features - Time stamp external events - Period output signals configurable from user space - Synchronization of the Linux system time via the PPS subsystem

79.1 PTP hardware clock kernel API

A PTP clock driver registers itself with the class driver. The class driver handles all of the dealings with user space. The author of a clock driver need only implement the details of programming the clock hardware. The clock driver notifies the class driver of asynchronous events (alarms and external time stamps) via a simple message passing interface.

The class driver supports multiple PTP clock drivers. In normal use cases, only one PTP clock is needed. However, for testing and development, it can be useful to have more than one clock in a single system, in order to allow performance comparisons.

79.2 PTP hardware clock user space API

The class driver also creates a character device for each registered clock. User space can use an open file descriptor from the character device as a POSIX clock id and may call clock_gettime, clock_settime, and clock_adjtime. These calls implement the basic clock operations.

User space programs may control the clock using standardized ioctls. A program may query, enable, configure, and disable the ancillary clock features. User space can receive time stamped events via blocking read() and poll().

79.3 Writing clock drivers

Clock drivers include include/linux/ptp_clock_kernel.h and register themselves by presenting a 'struct ptp_clock_info' to the registration method. Clock drivers must implement all of the functions in the interface. If a clock does not offer a particular ancillary feature, then the driver should just return -EOPNOTSUPP from those functions.

Drivers must ensure that all of the methods in interface are reentrant. Since most hardware implementations treat the time value as a 64 bit integer accessed as two 32 bit registers, drivers should use spin_lock_irqsave/spin_unlock_irqrestore to protect against concurrent access. This locking cannot be accomplished in class driver, since the lock may also be needed by the clock driver's interrupt service routine.

79.4 Supported hardware

- Freescale eTSEC gianfar
  - 2 Time stamp external triggers, programmable polarity (opt. interrupt)
  - 2 Alarm registers (optional interrupt)
  - 3 Periodic signals (optional interrupt)
- National DP83640
  - 6 GPIOs programmable as inputs or outputs
  - 6 GPIOs with dedicated functions (LED/JTAG/clock) can also be used as general inputs or outputs
  - GPIO inputs can time stamp external triggers
  - GPIO outputs can produce periodic signals
  - 1 interrupt pin
- Intel IXP465
  - Auxiliary Slave/Master Mode Snapshot (optional interrupt)
  - Target Time (optional interrupt)

CHAPTER

EIGHTY

GENERIC PHY FRAMEWORK

80.1 PHY subsystem

Author Kishon Vijay Abraham I <kishon@ti.com>

This document explains the Generic PHY Framework along with the APIs provided, and how-to-use.

80.1.1 Introduction

PHY is the abbreviation for physical layer. It is used to connect a device to the physical medium e.g., the USB controller has a PHY to provide functions such as serialization, de-serialization, encoding, decoding and is responsible for obtaining the required data transmission rate. Note that some USB controllers have PHY functionality embedded into it and others use an external PHY. Other peripherals that use PHY include Wireless LAN, Ethernet, SATA etc.

The intention of creating this framework is to bring the PHY drivers spread all over the Linux kernel to drivers/phy to increase code re-use and for better code maintainability.

This framework will be of use only to devices that use external PHY (PHY functionality is not embedded within the controller).

80.1.2 Registering/Unregistering the PHY provider

PHY provider refers to an entity that implements one or more PHY instances. For the simple case where the PHY provider implements only a single instance of the PHY, the framework provides its own implementation of of_xlate in of_phy_simple_xlate. If the PHY provider implements multiple instances, it should provide its own implementation of of_xlate. of_xlate is used only for dt boot case.

```
#define of_phy_provider_register(dev, xlate)    \
        __of_phy_provider_register((dev), NULL, THIS_MODULE, (xlate))

#define devm_of_phy_provider_register(dev, xlate)       \
        __devm_of_phy_provider_register((dev), NULL, THIS_MODULE,
                                        (xlate))
```

of_phy_provider_register and devm_of_phy_provider_register macros can be used to register the phy_provider and it takes device and of_xlate as arguments. For the

dt boot case, all PHY providers should use one of the above 2 macros to register the PHY provider.

Often the device tree nodes associated with a PHY provider will contain a set of children that each represent a single PHY. Some bindings may nest the child nodes within extra levels for context and extensibility, in which case the low level of_phy_provider_register_full() and devm_of_phy_provider_register_full() macros can be used to override the node containing the children.

```
#define of_phy_provider_register_full(dev, children, xlate) \
        __of_phy_provider_register(dev, children, THIS_MODULE, xlate)

#define devm_of_phy_provider_register_full(dev, children, xlate) \
        __devm_of_phy_provider_register_full(dev, children,
                                             THIS_MODULE, xlate)

void devm_of_phy_provider_unregister(struct device *dev,
        struct phy_provider *phy_provider);
void of_phy_provider_unregister(struct phy_provider *phy_provider);
```

devm_of_phy_provider_unregister and of_phy_provider_unregister can be used to unregister the PHY.

80.1.3 Creating the PHY

The PHY driver should create the PHY in order for other peripheral controllers to make use of it. The PHY framework provides 2 APIs to create the PHY.

```
struct phy *phy_create(struct device *dev, struct device_node *node,
                       const struct phy_ops *ops);
struct phy *devm_phy_create(struct device *dev,
                            struct device_node *node,
                            const struct phy_ops *ops);
```

The PHY drivers can use one of the above 2 APIs to create the PHY by passing the device pointer and phy ops. phy_ops is a set of function pointers for performing PHY operations such as init, exit, power_on and power_off.

Inorder to dereference the private data (in phy_ops), the phy provider driver can use phy_set_drvdata() after creating the PHY and use phy_get_drvdata() in phy_ops to get back the private data.

4. Getting a reference to the PHY

Before the controller can make use of the PHY, it has to get a reference to it. This framework provides the following APIs to get a reference to the PHY.

```
struct phy *phy_get(struct device *dev, const char *string);
struct phy *phy_optional_get(struct device *dev, const char *string);
struct phy *devm_phy_get(struct device *dev, const char *string);
struct phy *devm_phy_optional_get(struct device *dev,
                                  const char *string);
struct phy *devm_of_phy_get_by_index(struct device *dev,
                                     struct device_node *np,
                                     int index);
```

phy_get, phy_optional_get, devm_phy_get and devm_phy_optional_get can be used to get the PHY. In the case of dt boot, the string arguments should contain the phy name as given in the dt data and in the case of non-dt boot, it should contain the label of the PHY. The two devm_phy_get associates the device with the PHY using devres on successful PHY get. On driver detach, release function is invoked on the devres data and devres data is freed. phy_optional_get and devm_phy_optional_get should be used when the phy is optional. These two functions will never return -ENODEV, but instead returns NULL when the phy cannot be found.Some generic drivers, such as ehci, may use multiple phys and for such drivers referencing phy(s) by name(s) does not make sense. In this case, devm_of_phy_get_by_index can be used to get a phy reference based on the index.

It should be noted that NULL is a valid phy reference. All phy consumer calls on the NULL phy become NOPs. That is the release calls, the phy_init() and phy_exit() calls, and phy_power_on() and phy_power_off() calls are all NOP when applied to a NULL phy. The NULL phy is useful in devices for handling optional phy devices.

80.1.4 Releasing a reference to the PHY

When the controller no longer needs the PHY, it has to release the reference to the PHY it has obtained using the APIs mentioned in the above section. The PHY framework provides 2 APIs to release a reference to the PHY.

```
void phy_put(struct phy *phy);
void devm_phy_put(struct device *dev, struct phy *phy);
```

Both these APIs are used to release a reference to the PHY and devm_phy_put destroys the devres associated with this PHY.

80.1.5 Destroying the PHY

When the driver that created the PHY is unloaded, it should destroy the PHY it created using one of the following 2 APIs:

```
void phy_destroy(struct phy *phy);
void devm_phy_destroy(struct device *dev, struct phy *phy);
```

Both these APIs destroy the PHY and devm_phy_destroy destroys the devres associated with this PHY.

80.1.6 PM Runtime

This subsystem is pm runtime enabled. So while creating the PHY, pm_runtime_enable of the phy device created by this subsystem is called and while destroying the PHY, pm_runtime_disable is called. Note that the phy device created by this subsystem will be a child of the device that calls phy_create (PHY provider device).

So pm_runtime_get_sync of the phy_device created by this subsystem will invoke pm_runtime_get_sync of PHY provider device because of parent-child relationship. It should also be noted that phy_power_on and phy_power_off

performs phy_pm_runtime_get_sync and phy_pm_runtime_put respectively. There are exported APIs like phy_pm_runtime_get, phy_pm_runtime_get_sync, phy_pm_runtime_put, phy_pm_runtime_put_sync, phy_pm_runtime_allow and phy_pm_runtime_forbid for performing PM operations.

80.1.7 PHY Mappings

In order to get reference to a PHY without help from DeviceTree, the framework offers lookups which can be compared to clkdev that allow clk structures to be bound to devices. A lookup can be made during runtime when a handle to the struct phy already exists.

The framework offers the following API for registering and unregistering the lookups:

```
int phy_create_lookup(struct phy *phy, const char *con_id,
                      const char *dev_id);
void phy_remove_lookup(struct phy *phy, const char *con_id,
                       const char *dev_id);
```

80.1.8 DeviceTree Binding

The documentation for PHY dt binding can be found @ Documentation/devicetree/bindings/phy/phy-bindings.txt

80.2 Samsung USB 2.0 PHY adaptation layer

80.2.1 1. Description

The architecture of the USB 2.0 PHY module in Samsung SoCs is similar among many SoCs. In spite of the similarities it proved difficult to create a one driver that would fit all these PHY controllers. Often the differences were minor and were found in particular bits of the registers of the PHY. In some rare cases the order of register writes or the PHY powering up process had to be altered. This adaptation layer is a compromise between having separate drivers and having a single driver with added support for many special cases.

80.2.2 2. Files description

- **phy-samsung-usb2.c** This is the main file of the adaptation layer. This file contains the probe function and provides two callbacks to the Generic PHY Framework. This two callbacks are used to power on and power off the phy. They carry out the common work that has to be done on all version of the PHY module. Depending on which SoC was chosen they execute SoC specific callbacks. The specific SoC version is selected by choosing the appropriate compatible string. In addition, this file contains struct of_device_id definitions for particular SoCs.

- **phy-samsung-usb2.h** This is the include file. It declares the structures used by this driver. In addition it should contain extern declarations for structures that describe particular SoCs.

80.2.3 3. Supporting SoCs

To support a new SoC a new file should be added to the drivers/phy directory. Each SoC' s configuration is stored in an instance of the struct samsung_usb2_phy_config:

```
struct samsung_usb2_phy_config {
      const struct samsung_usb2_common_phy *phys;
      int (*rate_to_clk)(unsigned long, u32 *);
      unsigned int num_phys;
      bool has_mode_switch;
};
```

The num_phys is the number of phys handled by the driver. *phys is an array that contains the configuration for each phy. The has_mode_switch property is a boolean flag that determines whether the SoC has USB host and device on a single pair of pins. If so, a special register has to be modified to change the internal routing of these pins between a USB device or host module.

For example the configuration for Exynos 4210 is following:

```
const struct samsung_usb2_phy_config exynos4210_usb2_phy_config = {
      .has_mode_switch        = 0,
      .num_phys               = EXYNOS4210_NUM_PHYS,
      .phys                   = exynos4210_phys,
      .rate_to_clk            = exynos4210_rate_to_clk,
}
```

- int (*rate_to_clk)(unsigned long, u32 *)

  The rate_to_clk callback is to convert the rate of the clock used as the reference clock for the PHY module to the value that should be written in the hardware register.

The exynos4210_phys configuration array is as follows:

```
static const struct samsung_usb2_common_phy exynos4210_phys[] = {
      {
              .label          = "device",
              .id             = EXYNOS4210_DEVICE,
              .power_on       = exynos4210_power_on,
              .power_off      = exynos4210_power_off,
      },
      {
              .label          = "host",
              .id             = EXYNOS4210_HOST,
              .power_on       = exynos4210_power_on,
              .power_off      = exynos4210_power_off,
      },
      {
              .label          = "hsic0",
```

(continues on next page)

(continued from previous page)

```
                .id             = EXYNOS4210_HSIC0,
                .power_on       = exynos4210_power_on,
                .power_off      = exynos4210_power_off,
        },
        {
                .label          = "hsic1",
                .id             = EXYNOS4210_HSIC1,
                .power_on       = exynos4210_power_on,
                .power_off      = exynos4210_power_off,
        },
        {},
};
```

- int (*power_on)(struct samsung_usb2_phy_instance *); int (*power_off)(struct samsung_usb2_phy_instance *);

  These two callbacks are used to power on and power off the phy by modifying appropriate registers.

Final change to the driver is adding appropriate compatible value to the phy-samsung-usb2.c file. In case of Exynos 4210 the following lines were added to the struct of_device_id samsung_usb2_phy_of_match[] array:

```
#ifdef CONFIG_PHY_EXYNOS4210_USB2
        {
                .compatible = "samsung,exynos4210-usb2-phy",
                .data = &exynos4210_usb2_phy_config,
        },
#endif
```

To add further flexibility to the driver the Kconfig file enables to include support for selected SoCs in the compiled driver. The Kconfig entry for Exynos 4210 is following:

```
config PHY_EXYNOS4210_USB2
        bool "Support for Exynos 4210"
        depends on PHY_SAMSUNG_USB2
        depends on CPU_EXYNOS4210
        help
          Enable USB PHY support for Exynos 4210. This option requires that
          Samsung USB 2.0 PHY driver is enabled and means that support for␣
↪this
          particular SoC is compiled in the driver. In case of Exynos 4210␣
↪four
          phys are available - device, host, HSCI0 and HSCI1.
```

The newly created file that supports the new SoC has to be also added to the Makefile. In case of Exynos 4210 the added line is following:

```
obj-$(CONFIG_PHY_EXYNOS4210_USB2)       += phy-exynos4210-usb2.o
```

After completing these steps the support for the new SoC should be ready.

CHAPTER

EIGHTYONE

INTEL MID PTI

The Intel MID PTI project is HW implemented in Intel Atom system-on-a-chip designs based on the Parallel Trace Interface for MIPI P1149.7 cJTAG standard. The kernel solution for this platform involves the following files:

```
./include/linux/pti.h
./drivers/.../n_tracesink.h
./drivers/.../n_tracerouter.c
./drivers/.../n_tracesink.c
./drivers/.../pti.c
```

pti.c is the driver that enables various debugging features popular on platforms from certain mobile manufacturers. n_tracerouter.c and n_tracesink.c allow extra system information to be collected and routed to the pti driver, such as trace debugging data from a modem. Although n_tracerouter and n_tracesink are a part of the complete PTI solution, these two line disciplines can work separately from pti.c and route any data stream from one /dev/tty node to another /dev/tty node via kernel-space. This provides a stable, reliable connection that will not break unless the user-space application shuts down (plus avoids kernel->user->kernel context switch overheads of routing data).

An example debugging usage for this driver system:

- Hook /dev/ttyPTI0 to syslogd. Opening this port will also start a console device to further capture debugging messages to PTI.
- Hook /dev/ttyPTI1 to modem debugging data to write to PTI HW. This is where n_tracerouter and n_tracesink are used.
- Hook /dev/pti to a user-level debugging application for writing to PTI HW.
- Use mipi_ Kernel Driver API in other device drivers for debugging to PTI by first requesting a PTI write address via mipi_request_masterchannel(1).

Below is example pseudo-code on how a 'privileged' application can hook up n_tracerouter and n_tracesink to any tty on a system. 'Privileged' means the application has enough privileges to successfully manipulate the ldisc drivers but is not just blindly executing as 'root'. Keep in mind the use of ioctl(,TIOCSETD,) is not specific to the n_tracerouter and n_tracesink line discpline drivers but is a generic operation for a program to use a line discpline driver on a tty port other than the default n_tty:

```
////////// To hook up n_tracerouter and n_tracesink /////////
```

(continues on next page)

(continued from previous page)

```
// Note that n_tracerouter depends on n_tracesink.
#include <errno.h>
#define ONE_TTY "/dev/ttyOne"
#define TWO_TTY "/dev/ttyTwo"

// needed global to hand onto ldisc connection
static int g_fd_source = -1;
static int g_fd_sink  = -1;

// these two vars used to grab LDISC values from loaded ldisc drivers
// in OS.  Look at /proc/tty/ldiscs to get the right numbers from
// the ldiscs loaded in the system.
int source_ldisc_num, sink_ldisc_num = -1;
int retval;

g_fd_source = open(ONE_TTY, O_RDWR); // must be R/W
g_fd_sink   = open(TWO_TTY, O_RDWR); // must be R/W

if (g_fd_source <= 0) || (g_fd_sink <= 0) {
   // doubt you'll want to use these exact error lines of code
   printf("Error on open(). errno: %d\n",errno);
   return errno;
}

retval = ioctl(g_fd_sink, TIOCSETD, &sink_ldisc_num);
if (retval < 0) {
   printf("Error on ioctl().  errno: %d\n", errno);
   return errno;
}

retval = ioctl(g_fd_source, TIOCSETD, &source_ldisc_num);
if (retval < 0) {
   printf("Error on ioctl().  errno: %d\n", errno);
   return errno;
}

////////////// To disconnect n_tracerouter and n_tracesink ////////

// First make sure data through the ldiscs has stopped.

// Second, disconnect ldiscs.  This provides a
// little cleaner shutdown on tty stack.
sink_ldisc_num = 0;
source_ldisc_num = 0;
ioctl(g_fd_uart, TIOCSETD, &sink_ldisc_num);
ioctl(g_fd_gadget, TIOCSETD, &source_ldisc_num);

// Three, program closes connection, and cleanup:
close(g_fd_uart);
close(g_fd_gadget);
g_fd_uart = g_fd_gadget = NULL;
```

CHAPTER

EIGHTYTWO

PULSE WIDTH MODULATION (PWM) INTERFACE

This provides an overview about the Linux PWM interface

PWMs are commonly used for controlling LEDs, fans or vibrators in cell phones. PWMs with a fixed purpose have no need implementing the Linux PWM API (although they could). However, PWMs are often found as discrete devices on SoCs which have no fixed purpose. It's up to the board designer to connect them to LEDs or fans. To provide this kind of flexibility the generic PWM API exists.

82.1 Identifying PWMs

Users of the legacy PWM API use unique IDs to refer to PWM devices.

Instead of referring to a PWM device via its unique ID, board setup code should instead register a static mapping that can be used to match PWM consumers to providers, as given in the following example:

```
static struct pwm_lookup board_pwm_lookup[] = {
        PWM_LOOKUP("tegra-pwm", 0, "pwm-backlight", NULL,
                   50000, PWM_POLARITY_NORMAL),
};

static void __init board_init(void)
{
        ...
        pwm_add_table(board_pwm_lookup, ARRAY_SIZE(board_pwm_lookup));
        ...
}
```

82.2 Using PWMs

Legacy users can request a PWM device using `pwm_request()` and free it after usage with `pwm_free()`.

New users should use the `pwm_get()` function and pass to it the consumer device or a consumer name. `pwm_put()` is used to free the PWM device. Managed variants of these functions, `devm_pwm_get()` and `devm_pwm_put()`, also exist.

After being requested, a PWM has to be configured using:

```
int pwm_apply_state(struct pwm_device *pwm, struct pwm_state *state);
```

This API controls both the PWM period/duty_cycle config and the enable/disable state.

The `pwm_config()`, `pwm_enable()` and `pwm_disable()` functions are just wrappers around `pwm_apply_state()` and should not be used if the user wants to change several parameter at once. For example, if you see `pwm_config()` and pwm_{enable,disable}() calls in the same function, this probably means you should switch to `pwm_apply_state()`.

The PWM user API also allows one to query the PWM state with `pwm_get_state()`.

In addition to the PWM state, the PWM API also exposes PWM arguments, which are the reference PWM config one should use on this PWM. PWM arguments are usually platform-specific and allows the PWM user to only care about dutycycle relatively to the full period (like, duty = 50% of the period). struct pwm_args contains 2 fields (period and polarity) and should be used to set the initial PWM config (usually done in the probe function of the PWM user). PWM arguments are retrieved with pwm_get_args().

All consumers should really be reconfiguring the PWM upon resume as appropriate. This is the only way to ensure that everything is resumed in the proper order.

82.3 Using PWMs with the sysfs interface

If CONFIG_SYSFS is enabled in your kernel configuration a simple sysfs interface is provided to use the PWMs from userspace. It is exposed at /sys/class/pwm/. Each probed PWM controller/chip will be exported as pwmchipN, where N is the base of the PWM chip. Inside the directory you will find:

npwm The number of PWM channels this chip supports (read-only).

export Exports a PWM channel for use with sysfs (write-only).

unexport Unexports a PWM channel from sysfs (write-only).

The PWM channels are numbered using a per-chip index from 0 to npwm-1.

When a PWM channel is exported a pwmX directory will be created in the pwmchipN directory it is associated with, where X is the number of the channel that was exported. The following properties will then be available:

period The total period of the PWM signal (read/write). Value is in nanoseconds and is the sum of the active and inactive time of the PWM.

duty_cycle The active time of the PWM signal (read/write). Value is in nanoseconds and must be less than the period.

polarity Changes the polarity of the PWM signal (read/write). Writes to this property only work if the PWM chip supports changing the polarity. The polarity can only be changed if the PWM is not enabled. Value is the string “normal” or “inversed”.

enable Enable/disable the PWM signal (read/write).

- 0 - disabled
- 1 - enabled

82.4 Implementing a PWM driver

Currently there are two ways to implement pwm drivers. Traditionally there only has been the barebone API meaning that each driver has to implement the pwm_*() functions itself. This means that it's impossible to have multiple PWM drivers in the system. For this reason it's mandatory for new drivers to use the generic PWM framework.

A new PWM controller/chip can be added using `pwmchip_add()` and removed again with `pwmchip_remove()`. `pwmchip_add()` takes a filled in struct pwm_chip as argument which provides a description of the PWM chip, the number of PWM devices provided by the chip and the chip-specific implementation of the supported PWM operations to the framework.

When implementing polarity support in a PWM driver, make sure to respect the signal conventions in the PWM framework. By definition, normal polarity characterizes a signal starts high for the duration of the duty cycle and goes low for the remainder of the period. Conversely, a signal with inversed polarity starts low for the duration of the duty cycle and goes high for the remainder of the period.

Drivers are encouraged to implement ->apply() instead of the legacy ->enable(), ->disable() and ->config() methods. Doing that should provide atomicity in the PWM config workflow, which is required when the PWM controls a critical device (like a regulator).

The implementation of ->get_state() (a method used to retrieve initial PWM state) is also encouraged for the same reason: letting the PWM user know about the current PWM state would allow him to avoid glitches.

Drivers should not implement any power management. In other words, consumers should implement it as described in the "Using PWMs" section.

82.5 Locking

The PWM core list manipulations are protected by a mutex, so `pwm_request()` and `pwm_free()` may not be called from an atomic context. Currently the PWM core does not enforce any locking to `pwm_enable()`, `pwm_disable()` and `pwm_config()`, so the calling context is currently driver specific. This is an issue derived from the former barebone API and should be fixed soon.

82.6 Helpers

Currently a PWM can only be configured with period_ns and duty_ns. For several use cases freq_hz and duty_percent might be better. Instead of calculating this in your driver please consider adding appropriate helpers to the framework.

CHAPTER

EIGHTYTHREE

RFKILL - RF KILL SWITCH SUPPORT

Contents

83.1 Introduction

The rfkill subsystem provides a generic interface for disabling any radio transmitter in the system. When a transmitter is blocked, it shall not radiate any power.

The subsystem also provides the ability to react on button presses and disable all transmitters of a certain type (or all). This is intended for situations where transmitters need to be turned off, for example on aircraft.

The rfkill subsystem has a concept of "hard" and "soft" block, which differ little in their meaning (block == transmitters off) but rather in whether they can be changed or not:

- **hard block** read-only radio block that cannot be overridden by software
- **soft block** writable radio block (need not be readable) that is set by the system software.

The rfkill subsystem has two parameters, rfkill.default_state and rfkill.master_switch_mode, which are documented in admin-guide/kernel-parameters.rst.

83.2 Implementation details

The rfkill subsystem is composed of three main components:

- the rfkill core,
- the deprecated rfkill-input module (an input layer handler, being replaced by userspace policy code) and
- the rfkill drivers.

The rfkill core provides API for kernel drivers to register their radio transmitter with the kernel, methods for turning it on and off, and letting the system know about hardware-disabled states that may be implemented on the device.

The rfkill core code also notifies userspace of state changes, and provides ways for userspace to query the current states. See the "Userspace support" section below.

When the device is hard-blocked (either by a call to rfkill_set_hw_state() or from query_hw_block), set_block() will be invoked for additional software block, but drivers can ignore the method call since they can use the return value of the function rfkill_set_hw_state() to sync the software state instead of keeping track of calls to set_block(). In fact, drivers should use the return value of rfkill_set_hw_state() unless the hardware actually keeps track of soft and hard block separately.

83.3 Kernel API

Drivers for radio transmitters normally implement an rfkill driver.

Platform drivers might implement input devices if the rfkill button is just that, a button. If that button influences the hardware then you need to implement an rfkill driver instead. This also applies if the platform provides a way to turn on/off the transmitter(s).

For some platforms, it is possible that the hardware state changes during suspend/hibernation, in which case it will be necessary to update the rfkill core with the current state at resume time.

To create an rfkill driver, driver's Kconfig needs to have:

```
depends on RFKILL || !RFKILL
```

to ensure the driver cannot be built-in when rfkill is modular. The !RFKILL case allows the driver to be built when rfkill is not configured, in which case all rfkill API can still be used but will be provided by static inlines which compile to almost nothing.

Calling rfkill_set_hw_state() when a state change happens is required from rfkill drivers that control devices that can be hard-blocked unless they also assign the poll_hw_block() callback (then the rfkill core will poll the device). Don't do this unless you cannot get the event in any other way.

rfkill provides per-switch LED triggers, which can be used to drive LEDs according to the switch state (LED_FULL when blocked, LED_OFF otherwise).

83.4 Userspace support

The recommended userspace interface to use is /dev/rfkill, which is a misc character device that allows userspace to obtain and set the state of rfkill devices and sets of devices. It also notifies userspace about device addition and removal. The API is a simple read/write API that is defined in linux/rfkill.h, with one ioctl that allows turning off the deprecated input handler in the kernel for the transition period.

Except for the one ioctl, communication with the kernel is done via read() and write() of instances of 'struct rfkill_event'. In this structure, the soft and hard block are properly separated (unlike sysfs, see below) and userspace is able to get a consistent snapshot of all rfkill devices in the system. Also, it is possible to switch all rfkill drivers (or all drivers of a specified type) into a state which also updates the default state for hotplugged devices.

After an application opens /dev/rfkill, it can read the current state of all devices. Changes can be obtained by either polling the descriptor for hotplug or state change events or by listening for uevents emitted by the rfkill core framework.

Additionally, each rfkill device is registered in sysfs and emits uevents.

rfkill devices issue uevents (with an action of "change"), with the following environment variables set:

```
RFKILL_NAME
RFKILL_STATE
RFKILL_TYPE
```

The content of these variables corresponds to the "name", "state" and "type" sysfs files explained above.

For further details consult Documentation/ABI/stable/sysfs-class-rfkill.

CHAPTER

EIGHTYFOUR

SUPPORT FOR SERIAL DEVICES

84.1 Low Level Serial API

This document is meant as a brief overview of some aspects of the new serial driver. It is not complete, any questions you have should be directed to <rmk@arm.linux.org.uk>

The reference implementation is contained within amba-pl011.c.

84.1.1 Low Level Serial Hardware Driver

The low level serial hardware driver is responsible for supplying port information (defined by uart_port) and a set of control methods (defined by uart_ops) to the core serial driver. The low level driver is also responsible for handling interrupts for the port, and providing any console support.

84.1.2 Console Support

The serial core provides a few helper functions. This includes identifing the correct port structure (via uart_get_console) and decoding command line arguments (uart_parse_options).

There is also a helper function (uart_console_write) which performs a character by character write, translating newlines to CRLF sequences. Driver writers are recommended to use this function rather than implementing their own version.

84.1.3 Locking

It is the responsibility of the low level hardware driver to perform the necessary locking using port->lock. There are some exceptions (which are described in the uart_ops listing below.)

There are two locks. A per-port spinlock, and an overall semaphore.

From the core driver perspective, the port->lock locks the following data:

```
port->mctrl
port->icount
port->state->xmit.head (circ_buf->head)
port->state->xmit.tail (circ_buf->tail)
```

The low level driver is free to use this lock to provide any additional locking.

The port_sem semaphore is used to protect against ports being added/ removed or reconfigured at inappropriate times. Since v2.6.27, this semaphore has been the 'mutex' member of the tty_port struct, and commonly referred to as the port mutex.

84.1.4 uart_ops

The uart_ops structure is the main interface between serial_core and the hardware specific driver. It contains all the methods to control the hardware.

tx_empty(port) This function tests whether the transmitter fifo and shifter for the port described by 'port' is empty. If it is empty, this function should return TIOCSER_TEMT, otherwise return 0. If the port does not support this operation, then it should return TIOCSER_TEMT.

Locking: none.

Interrupts: caller dependent.

This call must not sleep

set_mctrl(port, mctrl) This function sets the modem control lines for port described by 'port' to the state described by mctrl. The relevant bits of mctrl are:

- TIOCM_RTS RTS signal.
- TIOCM_DTR DTR signal.
- TIOCM_OUT1 OUT1 signal.
- TIOCM_OUT2 OUT2 signal.
- TIOCM_LOOP Set the port into loopback mode.

If the appropriate bit is set, the signal should be driven active. If the bit is clear, the signal should be driven inactive.

Locking: port->lock taken.

Interrupts: locally disabled.

This call must not sleep

get_mctrl(port) Returns the current state of modem control inputs. The state of the outputs should not be returned, since the core keeps track of their state. The state information should include:

- TIOCM_CAR state of DCD signal
- TIOCM_CTS state of CTS signal
- TIOCM_DSR state of DSR signal
- TIOCM_RI state of RI signal

The bit is set if the signal is currently driven active. If the port does not support CTS, DCD or DSR, the driver should indicate that the

signal is permanently active. If RI is not available, the signal should not be indicated as active.

Locking: port->lock taken.

Interrupts: locally disabled.

This call must not sleep

stop_tx(port) Stop transmitting characters. This might be due to the CTS line becoming inactive or the tty layer indicating we want to stop transmission due to an XOFF character.

The driver should stop transmitting characters as soon as possible.

Locking: port->lock taken.

Interrupts: locally disabled.

This call must not sleep

start_tx(port) Start transmitting characters.

Locking: port->lock taken.

Interrupts: locally disabled.

This call must not sleep

throttle(port) Notify the serial driver that input buffers for the line discipline are close to full, and it should somehow signal that no more characters should be sent to the serial port. This will be called only if hardware assisted flow control is enabled.

Locking: serialized with .unthrottle() and termios modification by the tty layer.

unthrottle(port) Notify the serial driver that characters can now be sent to the serial port without fear of overrunning the input buffers of the line disciplines.

This will be called only if hardware assisted flow control is enabled.

Locking: serialized with .throttle() and termios modification by the tty layer.

send_xchar(port,ch) Transmit a high priority character, even if the port is stopped. This is used to implement XON/XOFF flow control and tcflow(). If the serial driver does not implement this function, the tty core will append the character to the circular buffer and then call start_tx() / stop_tx() to flush the data out.

Do not transmit if ch == ‘0’ (__DISABLED_CHAR).

Locking: none.

Interrupts: caller dependent.

stop_rx(port) Stop receiving characters; the port is in the process of being closed.

Locking: port->lock taken.

Interrupts: locally disabled.

This call must not sleep

enable_ms(port) Enable the modem status interrupts.

This method may be called multiple times. Modem status interrupts should be disabled when the shutdown method is called.

Locking: port->lock taken.

Interrupts: locally disabled.

This call must not sleep

break_ctl(port,ctl) Control the transmission of a break signal. If ctl is nonzero, the break signal should be transmitted. The signal should be terminated when another call is made with a zero ctl.

Locking: caller holds tty_port->mutex

startup(port) Grab any interrupt resources and initialise any low level driver state. Enable the port for reception. It should not activate RTS nor DTR; this will be done via a separate call to set_mctrl.

This method will only be called when the port is initially opened.

Locking: port_sem taken.

Interrupts: globally disabled.

shutdown(port) Disable the port, disable any break condition that may be in effect, and free any interrupt resources. It should not disable RTS nor DTR; this will have already been done via a separate call to set_mctrl.

Drivers must not access port->state once this call has completed.

This method will only be called when there are no more users of this port.

Locking: port_sem taken.

Interrupts: caller dependent.

flush_buffer(port) Flush any write buffers, reset any DMA state and stop any ongoing DMA transfers.

This will be called whenever the port->state->xmit circular buffer is cleared.

Locking: port->lock taken.

Interrupts: locally disabled.

This call must not sleep

set_termios(port,termios,oldtermios) Change the port parameters, including word length, parity, stop bits. Update read_status_mask and ignore_status_mask to indicate the types of events we are interested in receiving. Relevant termios->c_cflag bits are:

CSIZE

- word size

CSTOPB

- 2 stop bits

PARENB

- parity enable

PARODD

- odd parity (when PARENB is in force)

CREAD

- enable reception of characters (if not set, still receive characters from the port, but throw them away.

CRTSCTS

- if set, enable CTS status change reporting

CLOCAL

- if not set, enable modem status change reporting.

Relevant termios->c_iflag bits are:

INPCK

- enable frame and parity error events to be passed to the TTY layer.

BRKINT / PARMRK

- both of these enable break events to be passed to the TTY layer.

IGNPAR

- ignore parity and framing errors

IGNBRK

- ignore break errors, If IGNPAR is also set, ignore overrun errors as well.

The interaction of the iflag bits is as follows (parity error given as an example):

Parity error	INPCK	IGNPAR	
n/a	0	n/a	character received, marked as TTY_NORMAL
None	1	n/a	character received, marked as TTY_NORMAL
Yes	1	0	character received, marked as TTY_PARITY
Yes	1	1	character discarded

Other flags may be used (eg, xon/xoff characters) if your hardware supports hardware "soft" flow control.

Locking: caller holds tty_port->mutex

Interrupts: caller dependent.

This call must not sleep

set_ldisc(port,termios) Notifier for discipline change. See Documentation/driver-api/serial/tty.rst.

Locking: caller holds tty_port->mutex

pm(port,state,oldstate) Perform any power management related activities on the specified port. State indicates the new state (defined by enum uart_pm_state), oldstate indicates the previous state.

This function should not be used to grab any resources.

This will be called when the port is initially opened and finally closed, except when the port is also the system console. This will occur even if CONFIG_PM is not set.

Locking: none.

Interrupts: caller dependent.

type(port) Return a pointer to a string constant describing the specified port, or return NULL, in which case the string 'unknown' is substituted.

Locking: none.

Interrupts: caller dependent.

release_port(port) Release any memory and IO region resources currently in use by the port.

Locking: none.

Interrupts: caller dependent.

request_port(port) Request any memory and IO region resources required by the port. If any fail, no resources should be registered when this function returns, and it should return -EBUSY on failure.

Locking: none.

Interrupts: caller dependent.

config_port(port,type) Perform any autoconfiguration steps required for the port. type contains a bit mask of the required configuration. UART_CONFIG_TYPE indicates that the port requires detection and identification. port->type should be set to the type found, or PORT_UNKNOWN if no port was detected.

UART_CONFIG_IRQ indicates autoconfiguration of the interrupt signal, which should be probed using standard kernel autoprobing techniques. This is not necessary on platforms where ports have interrupts internally hard wired (eg, system on a chip implementations).

Locking: none.

Interrupts: caller dependent.

verify_port(port,serinfo) Verify the new serial port information contained within serinfo is suitable for this port type.

Locking: none.

Interrupts: caller dependent.

ioctl(port,cmd,arg) Perform any port specific IOCTLs. IOCTL commands must be defined using the standard numbering system found in <asm/ioctl.h>

Locking: none.

Interrupts: caller dependent.

poll_init(port) Called by kgdb to perform the minimal hardware initialization needed to support poll_put_char() and poll_get_char(). Unlike ->startup() this should not request interrupts.

Locking: tty_mutex and tty_port->mutex taken.

Interrupts: n/a.

poll_put_char(port,ch) Called by kgdb to write a single character directly to the serial port. It can and should block until there is space in the TX FIFO.

Locking: none.

Interrupts: caller dependent.

This call must not sleep

poll_get_char(port) Called by kgdb to read a single character directly from the serial port. If data is available, it should be returned; otherwise the function should return NO_POLL_CHAR immediately.

Locking: none.

Interrupts: caller dependent.

This call must not sleep

84.1.5 Other functions

uart_update_timeout(port,cflag,baud) Update the FIFO drain timeout, port->timeout, according to the number of bits, parity, stop bits and baud rate.

Locking: caller is expected to take port->lock

Interrupts: n/a

uart_get_baud_rate(port,termios,old,min,max) Return the numeric baud rate for the specified termios, taking account of the special 38400 baud “kludge” . The B0 baud rate is mapped to 9600 baud.

If the baud rate is not within min..max, then if old is non-NULL, the original baud rate will be tried. If that exceeds the min..max constraint, 9600 baud will be returned. termios will be updated to the baud rate in use.

Note: min..max must always allow 9600 baud to be selected.

Locking: caller dependent.

Interrupts: n/a

uart_get_divisor(port,baud) Return the divisor (baud_base / baud) for the specified baud rate, appropriately rounded.

If 38400 baud and custom divisor is selected, return the custom divisor instead.

Locking: caller dependent.

Interrupts: n/a

uart_match_port(port1,port2) This utility function can be used to determine whether two uart_port structures describe the same port.

Locking: n/a

Interrupts: n/a

uart_write_wakeup(port) A driver is expected to call this function when the number of characters in the transmit buffer have dropped below a threshold.

Locking: port->lock should be held.

Interrupts: n/a

uart_register_driver(drv) Register a uart driver with the core driver. We in turn register with the tty layer, and initialise the core driver per-port state.

drv->port should be NULL, and the per-port structures should be registered using uart_add_one_port after this call has succeeded.

Locking: none

Interrupts: enabled

uart_unregister_driver() Remove all references to a driver from the core driver. The low level driver must have removed all its ports via the `uart_remove_one_port()` if it registered them with `uart_add_one_port()`.

Locking: none

Interrupts: enabled

uart_suspend_port()

uart_resume_port()

uart_add_one_port()

uart_remove_one_port()

84.1.6 Other notes

It is intended some day to drop the 'unused' entries from uart_port, and allow low level drivers to register their own individual uart_port's with the core. This will allow drivers to use uart_port as a pointer to a structure containing both the uart_port entry with their own extensions, thus:

```
struct my_port {
        struct uart_port        port;
        int                     my_stuff;
};
```

84.1.7 Modem control lines via GPIO

Some helpers are provided in order to set/get modem control lines via GPIO.

mctrl_gpio_init(port, idx): This will get the {cts,rts,⋯}-gpios from device tree if they are present and request them, set direction etc, and return an allocated structure. devm_* functions are used, so there's no need to call mctrl_gpio_free(). As this sets up the irq handling make sure to not handle changes to the gpio input lines in your driver, too.

mctrl_gpio_free(dev, gpios): This will free the requested gpios in mctrl_gpio_init(). As devm_* functions are used, there's generally no need to call this function.

mctrl_gpio_to_gpiod(gpios, gidx) This returns the gpio_desc structure associated to the modem line index.

mctrl_gpio_set(gpios, mctrl): This will sets the gpios according to the mctrl state.

mctrl_gpio_get(gpios, mctrl): This will update mctrl with the gpios values.

mctrl_gpio_enable_ms(gpios): Enables irqs and handling of changes to the ms lines.

mctrl_gpio_disable_ms(gpios): Disables irqs and handling of changes to the ms lines.

84.2 The Lockronomicon

Your guide to the ancient and twisted locking policies of the tty layer and the warped logic behind them. Beware all ye who read on.

84.2.1 Line Discipline

Line disciplines are registered with tty_register_ldisc() passing the discipline number and the ldisc structure. At the point of registration the discipline must be ready to use and it is possible it will get used before the call returns success. If the call returns an error then it won't get called. Do not re-use ldisc numbers as they are part of the userspace ABI and writing over an existing ldisc will cause demons to eat your computer. After the return the ldisc data has been copied so you may free your own copy of the structure. You must not re-register over the top of the line discipline even with the same data or your computer again will be eaten by demons.

In order to remove a line discipline call tty_unregister_ldisc(). In ancient times this always worked. In modern times the function will return -EBUSY if the ldisc is currently in use. Since the ldisc referencing code manages the module counts this should not usually be a concern.

Heed this warning: the reference count field of the registered copies of the tty_ldisc structure in the ldisc table counts the number of lines using this discipline. The reference count of the tty_ldisc structure within a tty counts the number of active users of the ldisc at this instant. In effect it counts the number of threads of execution within an ldisc method (plus those about to enter and exit although this detail matters not).

84.2.2 Line Discipline Methods

TTY side interfaces

open()	Called when the line discipline is attached to the terminal. No other call into the line discipline for this tty will occur until it completes successfully. Should initialize any state needed by the ldisc, and set receive_room in the tty_struct to the maximum amount of data the line discipline is willing to accept from the driver with a single call to receive_buf(). Returning an error will prevent the ldisc from being attached. Can sleep.
close()	This is called on a terminal when the line discipline is being unplugged. At the point of execution no further users will enter the ldisc code for this tty. Can sleep.
hangup()	Called when the tty line is hung up. The line discipline should cease I/O to the tty. No further calls into the ldisc code will occur. The return value is ignored. Can sleep.
read()	(optional) A process requests reading data from the line. Multiple read calls may occur in parallel and the ldisc must deal with serialization issues. If not defined, the process will receive an EIO error. May sleep.
write()	(optional) A process requests writing data to the line. Multiple write calls are serialized by the tty layer for the ldisc. If not defined, the process will receive an EIO error. May sleep.
flush_buffer()	(optional) May be called at any point between open and close, and instructs the line discipline to empty its input buffer.
set_termios()	(optional) Called on termios structure changes. The caller passes the old termios data and the current data is in the tty. Called under the termios semaphore so allowed to sleep. Serialized against itself only.
poll()	(optional) Check the status for the poll/select calls. Multiple poll calls may occur in parallel. May sleep.
ioctl()	(optional) Called when an ioctl is handed to the tty layer that might be for the ldisc. Multiple ioctl calls may occur in parallel. May sleep.
compat_ioctl()	(optional) Called when a 32 bit ioctl is handed to the tty layer that might be for the ldisc. Multiple ioctl calls may occur in parallel. May sleep.

Driver Side Interfaces

receive_buf()	(optional) Called by the low-level driver to hand a buffer of received bytes to the ldisc for processing. The number of bytes is guaranteed not to exceed the current value of tty->receive_room. All bytes must be processed.
receive_buf2()	(optional) Called by the low-level driver to hand a buffer of received bytes to the ldisc for processing. Returns the number of bytes processed. If both receive_buf() and receive_buf2() are defined, receive_buf2() should be preferred.
write_wakeup()	May be called at any point between open and close. The TTY_DO_WRITE_WAKEUP flag indicates if a call is needed but always races versus calls. Thus the ldisc must be careful about setting order and to handle unexpected calls. Must not sleep. The driver is forbidden from calling this directly from the ->write call from the ldisc as the ldisc is permitted to call the driver write method from this function. In such a situation defer it.
dcd_change()	Report to the tty line the current DCD pin status changes and the relative timestamp. The timestamp cannot be NULL.

Driver Access

Line discipline methods can call the following methods of the underlying hardware driver through the function pointers within the tty->driver structure:

write()	Write a block of characters to the tty device. Returns the number of characters accepted. The character buffer passed to this method is already in kernel space.
put_char()	Queues a character for writing to the tty device. If there is no room in the queue, the character is ignored.
flush_chars()	(Optional) If defined, must be called after queueing characters with put_char() in order to start transmission.
write_room()	Returns the numbers of characters the tty driver will accept for queueing to be written.
ioctl()	Invoke device specific ioctl. Expects data pointers to refer to userspace. Returns ENOIOCTLCMD for unrecognized ioctl numbers.
set_termios()	Notify the tty driver that the device's termios settings have changed. New settings are in tty->termios. Previous settings should be passed in the "old" argument. The API is defined such that the driver should return the actual modes selected. This means that the driver function is responsible for modifying any bits in the request it cannot fulfill to indicate the actual modes being used. A device with no hardware capability for change (e.g. a USB dongle or virtual port) can provide NULL for this method.
throttle()	Notify the tty driver that input buffers for the line discipline are close to full, and it should somehow signal that no more characters should be sent to the tty.
unthrottle()	Notify the tty driver that characters can now be sent to the tty without fear of overrunning the input buffers of the line disciplines.
stop()	Ask the tty driver to stop outputting characters to the tty device.
start()	Ask the tty driver to resume sending characters to the tty device.
hangup()	Ask the tty driver to hang up the tty device.
break_ctl()	(Optional) Ask the tty driver to turn on or off BREAK status on the RS-232 port. If state is -1, then the BREAK status should be turned on; if state is 0, then BREAK should be turned off. If this routine is not implemented, use ioctls TIOCSBRK / TIOCCBRK instead.
wait_until_sent()	Waits until the device has written out all of the characters in its transmitter FIFO.
send_xchar()	Send a high-priority XON/XOFF character to the device.

Flags

Line discipline methods have access to tty->flags field containing the following interesting flags:

TTY_THROTTLED	Driver input is throttled. The ldisc should call tty->driver->unthrottle() in order to resume reception when it is ready to process more data.
TTY_DO_WRITE_WAKEUP	If set, causes the driver to call the ldisc's write_wakeup() method in order to resume transmission when it can accept more data to transmit.
TTY_IO_ERROR	If set, causes all subsequent userspace read/write calls on the tty to fail, returning -EIO.
TTY_OTHER_CLOSED	Device is a pty and the other side has closed.
TTY_NO_WRITE_SPLIT	Prevent driver from splitting up writes into smaller chunks.

Locking

Callers to the line discipline functions from the tty layer are required to take line discipline locks. The same is true of calls from the driver side but not yet enforced.

Three calls are now provided:

```
ldisc = tty_ldisc_ref(tty);
```

takes a handle to the line discipline in the tty and returns it. If no ldisc is currently attached or the ldisc is being closed and re-opened at this point then NULL is returned. While this handle is held the ldisc will not change or go away:

```
tty_ldisc_deref(ldisc)
```

Returns the ldisc reference and allows the ldisc to be closed. Returning the reference takes away your right to call the ldisc functions until you take a new reference:

```
ldisc = tty_ldisc_ref_wait(tty);
```

Performs the same function as tty_ldisc_ref except that it will wait for an ldisc change to complete and then return a reference to the new ldisc.

While these functions are slightly slower than the old code they should have minimal impact as most receive logic uses the flip buffers and they only need to take a reference when they push bits up through the driver.

A caution: The ldisc->open(), ldisc->close() and driver->set_ldisc functions are called with the ldisc unavailable. Thus tty_ldisc_ref will fail in this situation if used within these functions. Ldisc and driver code calling its own functions must be careful in this case.

84.2.3 Driver Interface

open()	Called when a device is opened. May sleep
close()	Called when a device is closed. At the point of return from this call the driver must make no further ldisc calls of any kind. May sleep
write()	Called to write bytes to the device. May not sleep. May occur in parallel in special cases. Because this includes panic paths drivers generally shouldn't try and do clever locking here.
put_char()	Stuff a single character onto the queue. The driver is guaranteed following up calls to flush_chars.
flush_chars()	Ask the kernel to write put_char queue
write_room()	Return the number of characters that can be stuffed into the port buffers without overflow (or less). The ldisc is responsible for being intelligent about multi-threading of write_room/write calls
ioctl()	Called when an ioctl may be for the driver
set_termios()	Called on termios change, serialized against itself by a semaphore. May sleep.
set_ldisc()	Notifier for discipline change. At the point this is done the discipline is not yet usable. Can now sleep (I think)
throttle()	Called by the ldisc to ask the driver to do flow control. Serialization including with unthrottle is the job of the ldisc layer.
unthrottle()	Called by the ldisc to ask the driver to stop flow control.
stop()	Ldisc notifier to the driver to stop output. As with throttle the serializations with start() are down to the ldisc layer.
start()	Ldisc notifier to the driver to start output.
hangup()	Ask the tty driver to cause a hangup initiated from the host side. [Can sleep ??]
break_ctl()	Send RS232 break. Can sleep. Can get called in parallel, driver must serialize (for now), and with write calls.
wait_until_sent()	Wait for characters to exit the hardware queue of the driver. Can sleep
send_xchar()	Send XON/XOFF and if possible jump the queue with it in order to get fast flow control responses. Cannot sleep ??

84.3 Serial drivers

84.3.1 Cyclades-Z notes

The Cyclades-Z must have firmware loaded onto the card before it will operate. This operation should be performed during system startup,

The firmware, loader program and the latest device driver code are available from Cyclades at

ftp://ftp.cyclades.com/pub/cyclades/cyclades-z/linux/

84.3.2 MOXA Smartio/Industio Family Device Driver Installation Guide

Note: This file is outdated. It needs some care in order to make it updated to Kernel 5.0 and upper

Date: 01/21/2008

1. Introduction

The Smartio/Industio/UPCI family Linux driver supports following multiport boards.

- **2 ports multiport board** CP-102U, CP-102UL, CP-102UF CP-132U-I, CP-132UL, CP-132, CP-132I, CP132S, CP-132IS, CI-132, CI-132I, CI-132IS, (C102H, C102HI, C102HIS, C102P, CP-102, CP-102S)
- **4 ports multiport board** CP-104EL, CP-104UL, CP-104JU, CP-134U, CP-134U-I, C104H/PCI, C104HS/PCI, CP-114, CP-114I, CP-114S, CP-114IS, CP-114UL, C104H, C104HS, CI-104J, CI-104JS, CI-134, CI-134I, CI-134IS, (C114HI, CT-114I, C104P), POS-104UL, CB-114, CB-134I
- **8 ports multiport board** CP-118EL, CP-168EL, CP-118U, CP-168U, C168H/PCI, C168H, C168HS, (C168P), CB-108

This driver and installation procedure have been developed upon Linux Kernel 2.4.x and 2.6.x. This driver supports Intel x86 hardware platform. In order to maintain compatibility, this version has also been properly tested with RedHat, Mandrake, Fedora and S.u.S.E Linux. However, if compatibility problem occurs, please contact Moxa at support@moxa.com.tw.

In addition to device driver, useful utilities are also provided in this version. They are:

- **msdiag** Diagnostic program for displaying installed Moxa Smartio/Industio boards.
- **msmon** Monitor program to observe data count and line status signals.
- **msterm A simple terminal program which is useful in testing serial** ports.
- **io-irq.exe** Configuration program to setup ISA boards. Please note that this program can only be executed under DOS.

All the drivers and utilities are published in form of source code under GNU General Public License in this version. Please refer to GNU General Public License announcement in each source code file for more detail.

In Moxa's Web sites, you may always find latest driver at http://www.moxa.com/.

This version of driver can be installed as Loadable Module (Module driver) or built-in into kernel (Static driver). You may refer to following installation procedure for suitable one. Before you install the driver, please refer to hardware installation procedure in the User's Manual.

We assume the user should be familiar with following documents.

- Serial-HOWTO
- Kernel-HOWTO

2. System Requirement

- Hardware platform: Intel x86 machine
- Kernel version: 2.4.x or 2.6.x
- gcc version 2.72 or later
- Maximum 4 boards can be installed in combination

3. Installation

3.1 Hardware installation

There are two types of buses, ISA and PCI, for Smartio/Industio family multiport board.

ISA board

You'll have to configure CAP address, I/O address, Interrupt Vector as well as IRQ before installing this driver. Please refer to hardware installation procedure in User's Manual before proceed any further. Please make sure the JP1 is open after the ISA board is set properly.

PCI/UPCI board

You may need to adjust IRQ usage in BIOS to avoid from IRQ conflict with other ISA devices. Please refer to hardware installation procedure in User's Manual in advance.

PCI IRQ Sharing

Each port within the same multiport board shares the same IRQ. Up to 4 Moxa Smartio/Industio PCI Family multiport boards can be installed together on one system and they can share the same IRQ.

3.2 Driver files

The driver file may be obtained from ftp, CD-ROM or floppy disk. The first step, anyway, is to copy driver file "mxser.tgz" into specified directory. e.g. /moxa. The execute commands as below:

```
# cd /
# mkdir moxa
# cd /moxa
# tar xvf /dev/fd0
```

or:

```
# cd /
# mkdir moxa
# cd /moxa
# cp /mnt/cdrom/<driver directory>/mxser.tgz .
# tar xvfz mxser.tgz
```

3.3 Device naming convention

You may find all the driver and utilities files in /moxa/mxser. Following installation procedure depends on the model you' d like to run the driver. If you prefer module driver, please refer to 3.4. If static driver is required, please refer to 3.5.

Dialin and callout port

This driver remains traditional serial device properties. There are two special file name for each serial port. One is dial-in port which is named "ttyMxx" . For callout port, the naming convention is "cumxx" .

Device naming when more than 2 boards installed

Naming convention for each Smartio/Industio multiport board is predefined as below.

Board Num.	Dial-in Port	Callout port
1st board	ttyM0 - ttyM7	cum0 - cum7
2nd board	ttyM8 - ttyM15	cum8 - cum15
3rd board	ttyM16 - ttyM23	cum16 - cum23
4th board	ttyM24 - ttym31	cum24 - cum31

Note: Under Kernel 2.6 and upper, the cum Device is Obsolete. So use ttyM* device instead.

Board sequence

This driver will activate ISA boards according to the parameter set in the driver. After all specified ISA board activated, PCI board will be installed in the system automatically driven. Therefore the board number is sorted by the CAP address of ISA boards. For PCI boards, their sequence will be after ISA boards and C168H/PCI has higher priority than C104H/PCI boards.

3.4 Module driver configuration

Module driver is easiest way to install. If you prefer static driver installation, please skip this paragraph.

———- Prepare to use the MOXA driver ————————–

3.4.1 Create tty device with correct major number

Before using MOXA driver, your system must have the tty devices which are created with driver's major number. We offer one shell script "msmknod" to simplify the procedure. This step is only needed to be executed once. But you still need to do this procedure when:

a. You change the driver's major number. Please refer the "3.7"section.

b. Your total installed MOXA boards number is changed. Maybe you add/delete one MOXA board.

c. You want to change the tty name. This needs to modify the shell script "msmknod"

The procedure is:

```
# cd /moxa/mxser/driver
# ./msmknod
```

This shell script will require the major number for dial-in device and callout device to create tty device. You also need to specify the total installed MOXA board number. Default major numbers for dial-in device and callout device are 30, 35. If you need to change to other number, please refer section "3.7" for more detailed procedure. Msmknod will delete any special files occupying the same device naming.

3.4.2 Build the MOXA driver and utilities

Before using the MOXA driver and utilities, you need compile the all the source code. This step is only need to be executed once. But you still re-compile the source code if you modify the source code. For example, if you change the driver’ s major number (see “3.7” section), then you need to do this step again.

Find “Makefile” in /moxa/mxser, then run

```
# make clean; make install
```

..note:

```
For Red Hat 9, Red Hat Enterprise Linux AS3/ES3/WS3 & Fedora␣
↪Core1:
# make clean; make installsp1

For Red Hat Enterprise Linux AS4/ES4/WS4:
# make clean; make installsp2
```

The driver files “mxser.o” and utilities will be properly compiled and copied to system directories respectively.

————- Load MOXA driver————–

3.4.3 Load the MOXA driver

```
# modprobe mxser <argument>
```

will activate the module driver. You may run “lsmod” to check if “mxser” is activated. If the MOXA board is ISA board, the <argument> is needed. Please refer to section “3.4.5” for more information.

————- Load MOXA driver on boot ————–

3.4.4 Load the mxser driver

For the above description, you may manually execute “modprobe mxser” to activate this driver and run “rmmod mxser” to remove it.

However, it’ s better to have a boot time configuration to eliminate manual operation. Boot time configuration can be achieved by rc file. We offer one “rc.mxser” file to simplify the procedure under “moxa/mxser/driver” .

But if you use ISA board, please modify the “modprobe …” command to add the argument (see “3.4.5” section). After modifying the rc.mxser, please try to execute “/moxa/mxser/driver/rc.mxser” manually to make sure the modification is ok. If any error encountered, please try to modify again. If the modification is completed, follow the below step.

Run following command for setting rc files:

```
# cd /moxa/mxser/driver
# cp ./rc.mxser /etc/rc.d
# cd /etc/rc.d
```

Check “rc.serial” is existed or not. If “rc.serial” doesn’t exist, create it by vi, run “chmod 755 rc.serial” to change the permission.

Add “/etc/rc.d/rc.mxser” in last line.

Reboot and check if moxa.o activated by “lsmod” command.

3.4.5. specify CAP address

If you’d like to drive Smartio/Industio ISA boards in the system, you’ll have to add parameter to specify CAP address of given board while activating “mxser.o”. The format for parameters are as follows.:

```
modprobe mxser ioaddr=0x???,0x???,0x???,0x???
                       |  |  |    |
                       |  |  |    +- 4th ISA board
                       |  |  +------ 3rd ISA board
                       |  +--------- 2nd ISA board
                       +-------------------1st ISA board
```

3.5 Static driver configuration for Linux kernel 2.4.x and 2.6.x

Note: To use static driver, you must install the linux kernel source package.

3.5.1 Backup the built-in driver in the kernel

```
# cd /usr/src/linux/drivers/char
# mv mxser.c mxser.c.old

For Red Hat 7.x user, you need to create link:
# cd /usr/src
# ln -s linux-2.4 linux
```

3.5.2 Create link

```
# cd /usr/src/linux/drivers/char
# ln -s /moxa/mxser/driver/mxser.c mxser.c
```

3.5.3 Add CAP address list for ISA boards.

For PCI boards user, please skip this step.

In module mode, the CAP address for ISA board is given by parameter. In static driver configuration, you'll have to assign it within driver's source code. If you will not install any ISA boards, you may skip to next portion. The instructions to modify driver source code are as below.

a. run:

```
# cd /moxa/mxser/driver
# vi mxser.c
```

b. Find the array mxserBoardCAP[] as below:

```
static int mxserBoardCAP[] = {0x00, 0x00, 0x00, 0x00};
```

c. Change the address within this array using vi. For example, to driver 2 ISA boards with CAP address 0x280 and 0x180 as 1st and 2nd board. Just to change the source code as follows:

```
static int mxserBoardCAP[] = {0x280, 0x180, 0x00, 0x00};
```

3.5.4 Setup kernel configuration

Configure the kernel:

```
# cd /usr/src/linux
# make menuconfig
```

You will go into a menu-driven system. Please select [Character devices][Non-standard serial port support], enable the [Moxa SmartIO support] driver with "[*]" for built-in (not "[M]"), then select [Exit] to exit this program.

3.5.5 Rebuild kernel

The following are for Linux kernel rebuilding, for your reference only.

For appropriate details, please refer to the Linux document:

a. Run the following commands:

```
cd /usr/src/linux
make clean              # take a few minutes
make dep                # take a few minutes
make bzImage            # take probably 10-20 minutes
make install            # copy boot image to correct␣
↪position
```

f. Please make sure the boot kernel (vmlinuz) is in the correct position.

g. If you use ‘lilo’ utility, you should check /etc/lilo.conf ‘image’ item specified the path which is the ‘vmlinuz’ path, or you will load wrong (or old) boot kernel image (vmlinuz). After checking /etc/lilo.conf, please run “lilo”.

   Note that if the result of “make bzImage” is ERROR, then you have to go back to Linux configuration Setup. Type “make menuconfig” in directory /usr/src/linux.

3.5.6 Make tty device and special file

:: # cd /moxa/mxser/driver # ./msmknod

3.5.7 Make utility

```
# cd /moxa/mxser/utility
# make clean; make install
```

3.5.8 Reboot

3.6 Custom configuration

Although this driver already provides you default configuration, you still can change the device name and major number. The instruction to change these parameters are shown as below.

a. Change Device name

   If you’d like to use other device names instead of default naming convention, all you have to do is to modify the internal code within the shell script “msmknod”. First, you have to open “msmknod” by vi. Locate each line contains “ttyM” and “cum” and change them to the device name you desired. “msmknod” creates the device names you need next time executed.

b. Change Major number

   If major number 30 and 35 had been occupied, you may have to select 2 free major numbers for this driver. There are 3 steps to change major numbers.

3.6.1 Find free major numbers

In /proc/devices, you may find all the major numbers occupied in the system. Please select 2 major numbers that are available. e.g. 40, 45.

3.6.2 Create special files

Run /moxa/mxser/driver/msmknod to create special files with specified major numbers.

3.6.3 Modify driver with new major number

Run vi to open /moxa/mxser/driver/mxser.c. Locate the line contains "MXSERMAJOR". Change the content as below:

```
      #define         MXSERMAJOR              40
      #define         MXSERCUMAJOR            45

3.6.4 Run "make clean; make install" in /moxa/mxser/driver.
```

3.7 Verify driver installation

You may refer to /var/log/messages to check the latest status log reported by this driver whenever it's activated.

4. Utilities

There are 3 utilities contained in this driver. They are msdiag, msmon and msterm. These 3 utilities are released in form of source code. They should be compiled into executable file and copied into /usr/bin.

Before using these utilities, please load driver (refer 3.4 & 3.5) and make sure you had run the "msmknod" utility.

msdiag - Diagnostic

This utility provides the function to display what Moxa Smartio/Industio board found by driver in the system.

msmon - Port Monitoring

This utility gives the user a quick view about all the MOXA ports' activities. One can easily learn each port' s total received/transmitted (Rx/Tx) character count since the time when the monitoring is started.

Rx/Tx throughputs per second are also reported in interval basis (e.g. the last 5 seconds) and in average basis (since the time the monitoring is started). You can reset all ports' count by <HOME> key. <+> <-> (plus/minus) keys to change the displaying time interval. Press <ENTER> on the port, that cursor stay, to view the port' s communication parameters, signal status, and input/output queue.

msterm - Terminal Emulation

This utility provides data sending and receiving ability of all tty ports, especially for MOXA ports. It is quite useful for testing simple application, for example, sending AT command to a modem connected to the port or used as a terminal for login purpose. Note that this is only a dumb terminal emulation without handling full screen operation.

5. Setserial

Supported Setserial parameters are listed as below.

uart	set UART type(16450–>disable FIFO, 16550A–>enable FIFO)
close_delay	set the amount of time(in 1/100 of a second) that DTR should be kept low while being closed.
closing_wait	set the amount of time(in 1/100 of a second) that the serial port should wait for data to be drained while being closed, before the receiver is disable.
spd_hi	Use 57.6kb when the application requests 38.4kb.
spd_vhi	Use 115.2kb when the application requests 38.4kb.
spd_shi	Use 230.4kb when the application requests 38.4kb.
spd_warp	Use 460.8kb when the application requests 38.4kb.
spd_normal	Use 38.4kb when the application requests 38.4kb.
spd_cust	Use the custom divisor to set the speed when the application requests 38.4kb.
divisor	This option set the custom division.
baud_base	This option set the base baud rate.

6. Troubleshooting

The boot time error messages and solutions are stated as clearly as possible. If all the possible solutions fail, please contact our technical support team to get more help.

Error msg: More than 4 Moxa Smartio/Industio family boards found. Fifth board and after are ignored.

Solution: To avoid this problem, please unplug fifth and after board, because Moxa driver supports up to 4 boards.

Error msg: Request_irq fail, IRQ(?) may be conflict with another device.

Solution: Other PCI or ISA devices occupy the assigned IRQ. If you are not sure which device causes the situation, please check /proc/interrupts to find free IRQ and simply change another free IRQ for Moxa board.

Error msg: Board #: C1xx Series(CAP=xxx) interrupt number invalid.

Solution: Each port within the same multiport board shares the same IRQ. Please set one IRQ (IRQ doesn't equal to zero) for one Moxa board.

Error msg: No interrupt vector be set for Moxa ISA board(CAP=xxx).

Solution: Moxa ISA board needs an interrupt vector.Please refer to user's manual "Hardware Installation" chapter to set interrupt vector.

Error msg: Couldn't install MOXA Smartio/Industio family driver!

Solution: Load Moxa driver fail, the major number may conflict with other devices. Please refer to previous section 3.7 to change a free major number for Moxa driver.

Error msg: Couldn't install MOXA Smartio/Industio family callout driver!

Solution: Load Moxa callout driver fail, the callout device major number may conflict with other devices. Please refer to previous section 3.7 to change a free callout device major number for Moxa driver.

84.3.3 GSM 0710 tty multiplexor HOWTO

This line discipline implements the GSM 07.10 multiplexing protocol detailed in the following 3GPP document:

http://www.3gpp.org/ftp/Specs/archive/07_series/07.10/0710-720.zip

This document give some hints on how to use this driver with GPRS and 3G modems connected to a physical serial port.

How to use it

1. initialize the modem in 0710 mux mode (usually AT+CMUX= command) through its serial port. Depending on the modem used, you can pass more or less parameters to this command,
2. switch the serial line to using the n_gsm line discipline by using TIOCSETD ioctl,
3. configure the mux using GSMIOC_GETCONF / GSMIOC_SETCONF ioctl,
4. obtain base gsmtty number for the used serial port,

Major parts of the initialization program : (a good starting point is util-linux-ng/sys-utils/ldattach.c):

```
#include <stdio.h>
#include <stdint.h>
#include <linux/gsmmux.h>
#include <linux/tty.h>
#define DEFAULT_SPEED B115200
#define SERIAL_PORT   /dev/ttyS0

      int ldisc = N_GSM0710;
      struct gsm_config c;
      struct termios configuration;
      uint32_t first;

      /* open the serial port connected to the modem */
      fd = open(SERIAL_PORT, O_RDWR | O_NOCTTY | O_NDELAY);

      /* configure the serial port : speed, flow control ... */

      /* send the AT commands to switch the modem to CMUX mode
         and check that it's successful (should return OK) */
      write(fd, "AT+CMUX=0\r", 10);

      /* experience showed that some modems need some time before
         being able to answer to the first MUX packet so a delay
         may be needed here in some case */
      sleep(3);

      /* use n_gsm line discipline */
      ioctl(fd, TIOCSETD, &ldisc);

      /* get n_gsm configuration */
      ioctl(fd, GSMIOC_GETCONF, &c);
      /* we are initiator and need encoding 0 (basic) */
      c.initiator = 1;
      c.encapsulation = 0;
      /* our modem defaults to a maximum size of 127 bytes */
      c.mru = 127;
      c.mtu = 127;
      /* set the new configuration */
      ioctl(fd, GSMIOC_SETCONF, &c);
      /* get first gsmtty device node */
      ioctl(fd, GSMIOC_GETFIRST, &first);
      printf("first muxed line: /dev/gsmtty%i\n", first);
```

(continues on next page)

(continued from previous page)

```
    /* and wait for ever to keep the line discipline enabled */
    daemon(0,0);
    pause();
```

5. use these devices as plain serial ports.

   for example, it' s possible:

   - and to use gnokii to send / receive SMS on ttygsm1
   - to use ppp to establish a datalink on ttygsm2

6. first close all virtual ports before closing the physical port.

   Note that after closing the physical port the modem is still in multiplexing mode. This may prevent a successful re-opening of the port later. To avoid this situation either reset the modem if your hardware allows that or send a disconnect command frame manually before initializing the multiplexing mode for the second time. The byte sequence for the disconnect command frame is:

   ```
   0xf9, 0x03, 0xef, 0x03, 0xc3, 0x16, 0xf9.
   ```

Additional Documentation

More practical details on the protocol and how it's supported by industrial modems can be found in the following documents :

- http://www.telit.com/module/infopool/download.php?id=616
- http://www.u-blox.com/images/downloads/Product_Docs/LEON-G100-G200-MuxImplementation_ApplicationNote_%28GSM%20G1-CS-10002%29.pdf
- http://www.sierrawireless.com/Support/Downloads/AirPrime/WMP_Series/~/media/Support_Downloads/AirPrime/Application_notes/CMUX_Feature_Application_Note-Rev004.ashx
- http://wm.sim.com/sim/News/photo/2010721161442.pdf

11-03-08 - Eric Bénard - <eric@eukrea.com>

84.3.4 Comtrol(tm) RocketPort(R)/RocketModem(TM) Series

Device Driver for the Linux Operating System

Product overview

This driver provides a loadable kernel driver for the Comtrol RocketPort and RocketModem PCI boards. These boards provide, 2, 4, 8, 16, or 32 high-speed serial ports or modems. This driver supports up to a combination of four RocketPort or RocketModems boards in one machine simultaneously. This file assumes that you are using the RocketPort driver which is integrated into the kernel sources.

The driver can also be installed as an external module using the usual "make;make install" routine. This external module driver, obtainable from the Comtrol website listed below, is useful for updating the driver or installing it into kernels which do not have the driver configured into them. Installations instructions for the external module are in the included README and HW_INSTALL files.

RocketPort ISA and RocketModem II PCI boards currently are only supported by this driver in module form.

The RocketPort ISA board requires I/O ports to be configured by the DIP switches on the board. See the section "ISA Rocketport Boards" below for information on how to set the DIP switches.

You pass the I/O port to the driver using the following module parameters:

board1: I/O port for the first ISA board

board2: I/O port for the second ISA board

board3: I/O port for the third ISA board

board4: I/O port for the fourth ISA board

There is a set of utilities and scripts provided with the external driver (downloadable from http://www.comtrol.com) that ease the configuration and setup of the ISA cards.

The RocketModem II PCI boards require firmware to be loaded into the card before it will function. The driver has only been tested as a module for this board.

Installation Procedures

RocketPort/RocketModem PCI cards require no driver configuration, they are automatically detected and configured.

The RocketPort driver can be installed as a module (recommended) or built into the kernel. This is selected, as for other drivers, through the make config command from the root of the Linux source tree during the kernel build process.

The RocketPort/RocketModem serial ports installed by this driver are assigned device major number 46, and will be named /dev/ttyRx, where x is the port number starting at zero (ex. /dev/ttyR0, /devttyR1, ⋯). If you have multiple cards installed in the system, the mapping of port names to serial ports is displayed in the system log at /var/log/messages.

If installed as a module, the module must be loaded. This can be done manually by entering "modprobe rocket". To have the module loaded automatically upon system boot, edit a /etc/modprobe.d/*.conf file and add the line "alias char-major-46 rocket".

In order to use the ports, their device names (nodes) must be created with mknod. This is only required once, the system will retain the names once created. To create the RocketPort/RocketModem device names, use the command "mknod /dev/ttyRx c 46 x" where x is the port number starting at zero.

For example:

```
> mknod /dev/ttyR0 c 46 0
> mknod /dev/ttyR1 c 46 1
> mknod /dev/ttyR2 c 46 2
```

The Linux script MAKEDEV will create the first 16 ttyRx device names (nodes) for you:

```
>/dev/MAKEDEV ttyR
```

ISA Rocketport Boards

You must assign and configure the I/O addresses used by the ISA Rocketport card before installing and using it. This is done by setting a set of DIP switches on the Rocketport board.

Setting the I/O address

Before installing RocketPort(R) or RocketPort RA boards, you must find a range of I/O addresses for it to use. The first RocketPort card requires a 68-byte contiguous block of I/O addresses, starting at one of the following: 0x100h, 0x140h, 0x180h, 0x200h, 0x240h, 0x280h, 0x300h, 0x340h, 0x380h. This I/O address must be reflected in the DIP switches of all of the Rocketport cards.

The second, third, and fourth RocketPort cards require a 64-byte contiguous block of I/O addresses, starting at one of the following I/O addresses: 0x100h, 0x140h, 0x180h, 0x1C0h, 0x200h, 0x240h, 0x280h, 0x2C0h, 0x300h, 0x340h, 0x380h, 0x3C0h. The I/O address used by the second, third, and fourth Rocketport cards (if present) are set via software control. The DIP switch settings for the I/O address must be set to the value of the first Rocketport cards.

In order to distinguish each of the card from the others, each card must have a unique board ID set on the dip switches. The first Rocketport board must be set with the DIP switches corresponding to the first board, the second board must be set with the DIP switches corresponding to the second board, etc. IMPORTANT: The board ID is the only place where the DIP switch settings should differ between the various Rocketport boards in a system.

The I/O address range used by any of the RocketPort cards must not conflict with any other cards in the system, including other RocketPort cards. Below, you will find a list of commonly used I/O address ranges which may be in use by other devices in your system. On a Linux system, “cat /proc/ioports” will also be helpful in identifying what I/O addresses are being used by devices on your system.

Remember, the FIRST RocketPort uses 68 I/O addresses. So, if you set it for 0x100, it will occupy 0x100 to 0x143. This would mean that you CAN NOT set the second, third or fourth board for address 0x140 since the first 4 bytes of that range are used by the first board. You would need to set the second, third, or fourth board to one of the next available blocks such as 0x180.

RocketPort and RocketPort RA SW1 Settings:

```
          +-------------------------------+
          | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
          +-------+-------+---------------+
          | Unused| Card  | I/O Port Block|
          +-------------------------------+

DIP Switches                                DIP Switches
7    8                                      6    5
====================                        ====================
On   On   UNUSED, MUST BE ON.               On   On   First Card    <====␣
↪Default
                                            On   Off  Second Card
                                            Off  On   Third Card
                                            Off  Off  Fourth Card

DIP Switches          I/O Address Range
4    3    2    1     Used by the First Card
=====================================
On   Off  On   Off   100-143
On   Off  Off  On    140-183
On   Off  Off  Off   180-1C3       <==== Default
Off  On   On   Off   200-243
Off  On   Off  On    240-283
Off  On   Off  Off   280-2C3
Off  Off  On   Off   300-343
Off  Off  Off  On    340-383
Off  Off  Off  Off   380-3C3
```

Reporting Bugs

For technical support, please provide the following information: Driver version, kernel release, distribution of kernel, and type of board you are using. Error messages and log printouts port configuration details are especially helpful.

USA:

Phone

(612) 494-4100

FAX

(612) 494-4199

email support@comtrol.com

Comtrol Europe:

Phone +44 (0) 1 869 323-220

FAX +44 (0) 1 869 323-211

email support@comtrol.co.uk

Web: http://www.comtrol.com FTP: ftp.comtrol.com

84.3.5 ISO7816 Serial Communications

1. Introduction

ISO/IEC7816 is a series of standards specifying integrated circuit cards (ICC) also known as smart cards.

2. Hardware-related considerations

Some CPUs/UARTs (e.g., Microchip AT91) contain a built-in mode capable of handling communication with a smart card.

For these microcontrollers, the Linux driver should be made capable of working in both modes, and proper ioctls (see later) should be made available at user-level to allow switching from one mode to the other, and vice versa.

3. Data Structures Already Available in the Kernel

The Linux kernel provides the serial_iso7816 structure (see [1]) to handle ISO7816 communications. This data structure is used to set and configure ISO7816 parameters in ioctls.

Any driver for devices capable of working both as RS232 and ISO7816 should implement the iso7816_config callback in the uart_port structure. The serial_core calls iso7816_config to do the device specific part in response to TIOCGISO7816 and TIOCSISO7816 ioctls (see below). The iso7816_config callback receives a pointer to struct serial_iso7816.

4. Usage from user-level

From user-level, ISO7816 configuration can be get/set using the previous ioctls. For instance, to set ISO7816 you can use the following code:

```
#include <linux/serial.h>

/* Include definition for ISO7816 ioctls: TIOCSISO7816 and␣
↪TIOCGISO7816 */
#include <sys/ioctl.h>

/* Open your specific device (e.g., /dev/mydevice): */
int fd = open ("/dev/mydevice", O_RDWR);
if (fd < 0) {
        /* Error handling. See errno. */
}

struct serial_iso7816 iso7816conf;

/* Reserved fields as to be zeroed */
memset(&iso7816conf, 0, sizeof(iso7816conf));

/* Enable ISO7816 mode: */
```

(continues on next page)

(continued from previous page)

```
iso7816conf.flags |= SER_ISO7816_ENABLED;

/* Select the protocol: */
/* T=0 */
iso7816conf.flags |= SER_ISO7816_T(0);
/* or T=1 */
iso7816conf.flags |= SER_ISO7816_T(1);

/* Set the guard time: */
iso7816conf.tg = 2;

/* Set the clock frequency*/
iso7816conf.clk = 3571200;

/* Set transmission factors: */
iso7816conf.sc_fi = 372;
iso7816conf.sc_di = 1;

if (ioctl(fd_usart, TIOCSISO7816, &iso7816conf) < 0) {
        /* Error handling. See errno. */
}

/* Use read() and write() syscalls here... */

/* Close the device when finished: */
if (close (fd) < 0) {
        /* Error handling. See errno. */
}
```

5. References

[1] include/uapi/linux/serial.h

84.3.6 RS485 Serial Communications

1. Introduction

EIA-485, also known as TIA/EIA-485 or RS-485, is a standard defining the electrical characteristics of drivers and receivers for use in balanced digital multipoint systems. This standard is widely used for communications in industrial automation because it can be used effectively over long distances and in electrically noisy environments.

2. Hardware-related Considerations

Some CPUs/UARTs (e.g., Atmel AT91 or 16C950 UART) contain a built-in half-duplex mode capable of automatically controlling line direction by toggling RTS or DTR signals. That can be used to control external half-duplex hardware like an RS485 transceiver or any RS232-connected half-duplex devices like some modems.

For these microcontrollers, the Linux driver should be made capable of working in both modes, and proper ioctls (see later) should be made available at user-level to allow switching from one mode to the other, and vice versa.

3. Data Structures Already Available in the Kernel

The Linux kernel provides the serial_rs485 structure (see [1]) to handle RS485 communications. This data structure is used to set and configure RS485 parameters in the platform data and in ioctls.

The device tree can also provide RS485 boot time parameters (see [2] for bindings). The driver is in charge of filling this data structure from the values given by the device tree.

Any driver for devices capable of working both as RS232 and RS485 should implement the rs485_config callback in the uart_port structure. The serial_core calls rs485_config to do the device specific part in response to TIOCSRS485 and TIOCGRS485 ioctls (see below). The rs485_config callback receives a pointer to struct serial_rs485.

4. Usage from user-level

From user-level, RS485 configuration can be get/set using the previous ioctls. For instance, to set RS485 you can use the following code:

```
#include <linux/serial.h>

/* Include definition for RS485 ioctls: TIOCGRS485 and TIOCSRS485␣
↪*/
#include <sys/ioctl.h>

/* Open your specific device (e.g., /dev/mydevice): */
int fd = open ("/dev/mydevice", O_RDWR);
if (fd < 0) {
        /* Error handling. See errno. */
}

struct serial_rs485 rs485conf;

/* Enable RS485 mode: */
rs485conf.flags |= SER_RS485_ENABLED;

/* Set logical level for RTS pin equal to 1 when sending: */
rs485conf.flags |= SER_RS485_RTS_ON_SEND;
```

(continues on next page)

(continued from previous page)

```
/* or, set logical level for RTS pin equal to 0 when sending: */
rs485conf.flags &= ~(SER_RS485_RTS_ON_SEND);

/* Set logical level for RTS pin equal to 1 after sending: */
rs485conf.flags |= SER_RS485_RTS_AFTER_SEND;
/* or, set logical level for RTS pin equal to 0 after sending: */
rs485conf.flags &= ~(SER_RS485_RTS_AFTER_SEND);

/* Set rts delay before send, if needed: */
rs485conf.delay_rts_before_send = ...;

/* Set rts delay after send, if needed: */
rs485conf.delay_rts_after_send = ...;

/* Set this flag if you want to receive data even while sending␣
↪data */
rs485conf.flags |= SER_RS485_RX_DURING_TX;

if (ioctl (fd, TIOCSRS485, &rs485conf) < 0) {
        /* Error handling. See errno. */
}

/* Use read() and write() syscalls here... */

/* Close the device when finished: */
if (close (fd) < 0) {
        /* Error handling. See errno. */
}
```

5. References

[1] include/uapi/linux/serial.h

[2] Documentation/devicetree/bindings/serial/rs485.txt

CHAPTER

EIGHTYFIVE

SM501 DRIVER

The Silicon Motion SM501 multimedia companion chip is a multifunction device which may provide numerous interfaces including USB host controller USB gadget, asynchronous serial ports, audio functions, and a dual display video interface. The device may be connected by PCI or local bus with varying functions enabled.

85.1 Core

The core driver in drivers/mfd provides common services for the drivers which manage the specific hardware blocks. These services include locking for common registers, clock control and resource management.

The core registers drivers for both PCI and generic bus based chips via the platform device and driver system.

On detection of a device, the core initialises the chip (which may be specified by the platform data) and then exports the selected peripheral set as platform devices for the specific drivers.

The core re-uses the platform device system as the platform device system provides enough features to support the drivers without the need to create a new bus-type and the associated code to go with it.

85.2 Resources

Each peripheral has a view of the device which is implicitly narrowed to the specific set of resources that peripheral requires in order to function correctly.

The centralised memory allocation allows the driver to ensure that the maximum possible resource allocation can be made to the video subsystem as this is by-far the most resource-sensitive of the on-chip functions.

The primary issue with memory allocation is that of moving the video buffers once a display mode is chosen. Indeed when a video mode change occurs the memory footprint of the video subsystem changes.

Since video memory is difficult to move without changing the display (unless sufficient contiguous memory can be provided for the old and new modes simultaneously) the video driver fully utilises the memory area given to it by aligning fb0 to

the start of the area and fb1 to the end of it. Any memory left over in the middle is used for the acceleration functions, which are transient and thus their location is less critical as it can be moved.

85.3 Configuration

The platform device driver uses a set of platform data to pass configurations through to the core and the subsidiary drivers so that there can be support for more than one system carrying an SM501 built into a single kernel image.

The PCI driver assumes that the PCI card behaves as per the Silicon Motion reference design.

There is an errata (AB-5) affecting the selection of the of the M1XCLK and M1CLK frequencies. These two clocks must be sourced from the same PLL, although they can then be divided down individually. If this is not set, then SM501 may lock and hang the whole system. The driver will refuse to attach if the PLL selection is different.

CHAPTER

EIGHTYSIX

MSC KEYBOARD SCAN EXPANSION/GPIO EXPANSION DEVICE

86.1 What is smsc-ece1099?

The ECE1099 is a 40-Pin 3.3V Keyboard Scan Expansion or GPIO Expansion device. The device supports a keyboard scan matrix of 23x8. The device is connected to a Master via the SMSC BC-Link interface or via the SMBus. Keypad scan Input(KSI) and Keypad Scan Output(KSO) signals are multiplexed with GPIOs.

86.2 Interrupt generation

Interrupts can be generated by an edge detection on a GPIO pin or an edge detection on one of the bus interface pins. Interrupts can also be detected on the keyboard scan interface. The bus interrupt pin (BC_INT# or SMBUS_INT#) is asserted if any bit in one of the Interrupt Status registers is 1 and the corresponding Interrupt Mask bit is also 1.

In order for software to determine which device is the source of an interrupt, it should first read the Group Interrupt Status Register to determine which Status register group is a source for the interrupt. Software should read both the Status register and the associated Mask register, then AND the two values together. Bits that are 1 in the result of the AND are active interrupts. Software clears an interrupt by writing a 1 to the corresponding bit in the Status register.

86.3 Communication Protocol

- **SMbus slave Interface** The host processor communicates with the ECE1099 device through a series of read/write registers via the SMBus interface. SMBus is a serial communication protocol between a computer host and its peripheral devices. The SMBus data rate is 10KHz minimum to 400 KHz maximum
- **Slave Bus Interface** The ECE1099 device SMBus implementation is a subset of the SMBus interface to the host. The device is a slave-only SMBus device. The implementation in the device is a subset of SMBus since it only supports four protocols.

The Write Byte, Read Byte, Send Byte, and Receive Byte protocols are the only valid SMBus protocols for the device.

- **BC-LinkTM Interface** The BC-Link is a proprietary bus that allows communication between a Master device and a Companion device. The Master device uses this serial bus to read and write registers located on the Companion device. The bus comprises three signals, BC_CLK, BC_DAT and BC_INT#. The Master device always provides the clock, BC_CLK, and the Companion device is the source for an independent asynchronous interrupt signal, BC_INT#. The ECE1099 supports BC-Link speeds up to 24MHz.

CHAPTER

EIGHTYSEVEN

LINUX SWITCHTEC SUPPORT

Microsemi's "Switchtec" line of PCI switch devices is already supported by the kernel with standard PCI switch drivers. However, the Switchtec device advertises a special management endpoint which enables some additional functionality. This includes:

- Packet and Byte Counters
- Firmware Upgrades
- Event and Error logs
- Querying port link status
- Custom user firmware commands

The switchtec kernel module implements this functionality.

87.1 Interface

The primary means of communicating with the Switchtec management firmware is through the Memory-mapped Remote Procedure Call (MRPC) interface. Commands are submitted to the interface with a 4-byte command identifier and up to 1KB of command specific data. The firmware will respond with a 4-byte return code and up to 1KB of command-specific data. The interface only processes a single command at a time.

87.2 Userspace Interface

The MRPC interface will be exposed to userspace through a simple char device: /dev/switchtec#, one for each management endpoint in the system.

The char device has the following semantics:

- A write must consist of at least 4 bytes and no more than 1028 bytes. The first 4 bytes will be interpreted as the Command ID and the remainder will be used as the input data. A write will send the command to the firmware to begin processing.
- Each write must be followed by exactly one read. Any double write will produce an error and any read that doesn't follow a write will produce an error.

- A read will block until the firmware completes the command and return the 4-byte Command Return Value plus up to 1024 bytes of output data. (The length will be specified by the size parameter of the read call – reading less than 4 bytes will produce an error.)
- The poll call will also be supported for userspace applications that need to do other things while waiting for the command to complete.

The following IOCTLs are also supported by the device:

- SWITCHTEC_IOCTL_FLASH_INFO - Retrieve firmware length and number of partitions in the device.
- SWITCHTEC_IOCTL_FLASH_PART_INFO - Retrieve address and lengeth for any specified partition in flash.
- SWITCHTEC_IOCTL_EVENT_SUMMARY - Read a structure of bitmaps indicating all uncleared events.
- SWITCHTEC_IOCTL_EVENT_CTL - Get the current count, clear and set flags for any event. This ioctl takes in a switchtec_ioctl_event_ctl struct with the event_id, index and flags set (index being the partition or PFF number for non-global events). It returns whether the event has occurred, the number of times and any event specific data. The flags can be used to clear the count or enable and disable actions to happen when the event occurs. By using the SWITCHTEC_IOCTL_EVENT_FLAG_EN_POLL flag, you can set an event to trigger a poll command to return with POLLPRI. In this way, userspace can wait for events to occur.
- SWITCHTEC_IOCTL_PFF_TO_PORT and SWITCHTEC_IOCTL_PORT_TO_PFF convert between PCI Function Framework number (used by the event system) and Switchtec Logic Port ID and Partition number (which is more user friendly).

87.3 Non-Transparent Bridge (NTB) Driver

An NTB hardware driver is provided for the Switchtec hardware in ntb_hw_switchtec. Currently, it only supports switches configured with exactly 2 NT partitions and zero or more non-NT partitions. It also requires the following configuration settings:

- Both NT partitions must be able to access each other's GAS spaces. Thus, the bits in the GAS Access Vector under Management Settings must be set to support this.
- Kernel configuration MUST include support for NTB (CONFIG_NTB needs to be set)

NT EP BAR 2 will be dynamically configured as a Direct Window, and the configuration file does not need to configure it explicitly.

Please refer to Documentation/driver-api/ntb.rst in Linux source tree for an overall understanding of the Linux NTB stack. ntb_hw_switchtec works as an NTB Hardware Driver in this stack.

CHAPTER

EIGHTYEIGHT

SYNC FILE API GUIDE

Author Gustavo Padovan <gustavo at padovan dot org>

This document serves as a guide for device drivers writers on what the sync_file API is, and how drivers can support it. Sync file is the carrier of the fences(struct dma_fence) that are needed to synchronize between drivers or across process boundaries.

The sync_file API is meant to be used to send and receive fence information to/from userspace. It enables userspace to do explicit fencing, where instead of attaching a fence to the buffer a producer driver (such as a GPU or V4L driver) sends the fence related to the buffer to userspace via a sync_file.

The sync_file then can be sent to the consumer (DRM driver for example), that will not use the buffer for anything before the fence(s) signals, i.e., the driver that issued the fence is not using/processing the buffer anymore, so it signals that the buffer is ready to use. And vice-versa for the consumer -> producer part of the cycle.

Sync files allows userspace awareness on buffer sharing synchronization between drivers.

Sync file was originally added in the Android kernel but current Linux Desktop can benefit a lot from it.

88.1 in-fences and out-fences

Sync files can go either to or from userspace. When a sync_file is sent from the driver to userspace we call the fences it contains 'out-fences'. They are related to a buffer that the driver is processing or is going to process, so the driver creates an out-fence to be able to notify, through `dma_fence_signal()`, when it has finished using (or processing) that buffer. Out-fences are fences that the driver creates.

On the other hand if the driver receives fence(s) through a sync_file from userspace we call these fence(s) 'in-fences'. Receiving in-fences means that we need to wait for the fence(s) to signal before using any buffer related to the in-fences.

88.2 Creating Sync Files

When a driver needs to send an out-fence userspace it creates a sync_file.

Interface:

```
struct sync_file *sync_file_create(struct dma_fence *fence);
```

The caller pass the out-fence and gets back the sync_file. That is just the first step, next it needs to install an fd on sync_file->file. So it gets an fd:

```
fd = get_unused_fd_flags(O_CLOEXEC);
```

and installs it on sync_file->file:

```
fd_install(fd, sync_file->file);
```

The sync_file fd now can be sent to userspace.

If the creation process fail, or the sync_file needs to be released by any other reason fput(sync_file->file) should be used.

88.3 Receiving Sync Files from Userspace

When userspace needs to send an in-fence to the driver it passes file descriptor of the Sync File to the kernel. The kernel can then retrieve the fences from it.

Interface:

```
struct dma_fence *sync_file_get_fence(int fd);
```

The returned reference is owned by the caller and must be disposed of afterwards using `dma_fence_put()`. In case of error, a NULL is returned instead.

References:

1. struct sync_file in include/linux/sync_file.h
2. All interfaces mentioned above defined in include/linux/sync_file.h

CHAPTER

EIGHTYNINE

VFIO MEDIATED DEVICES

Author Neo Jia <cjia@nvidia.com>

Author Kirti Wankhede <kwankhede@nvidia.com>

89.1 Virtual Function I/O (VFIO) Mediated devices[1]

The number of use cases for virtualizing DMA devices that do not have built-in SR_IOV capability is increasing. Previously, to virtualize such devices, developers had to create their own management interfaces and APIs, and then integrate them with user space software. To simplify integration with user space software, we have identified common requirements and a unified management interface for such devices.

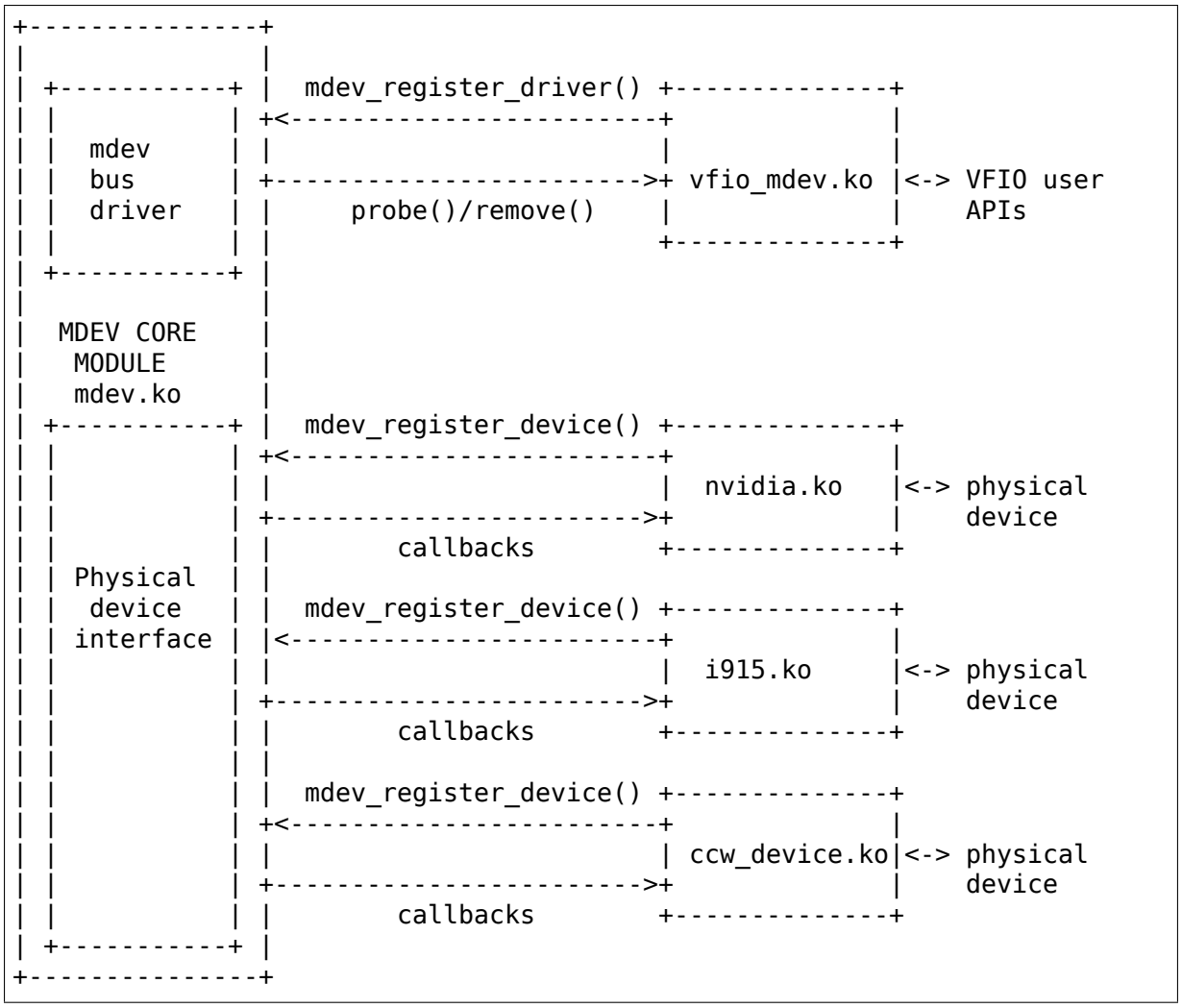
The VFIO driver framework provides unified APIs for direct device access. It is an IOMMU/device-agnostic framework for exposing direct device access to user space in a secure, IOMMU-protected environment. This framework is used for multiple devices, such as GPUs, network adapters, and compute accelerators. With direct device access, virtual machines or user space applications have direct access to the physical device. This framework is reused for mediated devices.

The mediated core driver provides a common interface for mediated device management that can be used by drivers of different devices. This module provides a generic interface to perform these operations:

- Create and destroy a mediated device
- Add a mediated device to and remove it from a mediated bus driver
- Add a mediated device to and remove it from an IOMMU group

The mediated core driver also provides an interface to register a bus driver. For example, the mediated VFIO mdev driver is designed for mediated devices and supports VFIO APIs. The mediated bus driver adds a mediated device to and removes it from a VFIO group.

The following high-level block diagram shows the main components and interfaces in the VFIO mediated driver framework. The diagram shows NVIDIA, Intel, and IBM devices as examples, as these devices are the first devices to use this module:



89.2 Registration Interfaces

The mediated core driver provides the following types of registration interfaces:

- Registration interface for a mediated bus driver
- Physical device driver interface

89.2.1 Registration Interface for a Mediated Bus Driver

The registration interface for a mediated bus driver provides the following structure to represent a mediated device's driver:

```
/*
 * struct mdev_driver [2] - Mediated device's driver
 * @name: driver name
 * @probe: called when new device created
 * @remove: called when device removed
 * @driver: device driver structure
 */
struct mdev_driver {
        const char *name;
        int  (*probe)  (struct device *dev);
        void (*remove) (struct device *dev);
        struct device_driver    driver;
};
```

A mediated bus driver for mdev should use this structure in the function calls to register and unregister itself with the core driver:

- Register:

```
extern int  mdev_register_driver(struct mdev_driver *drv,
                                 struct module *owner);
```

- Unregister:

```
extern void mdev_unregister_driver(struct mdev_driver *drv);
```

The mediated bus driver is responsible for adding mediated devices to the VFIO group when devices are bound to the driver and removing mediated devices from the VFIO when devices are unbound from the driver.

89.2.2 Physical Device Driver Interface

The physical device driver interface provides the mdev_parent_ops[3] structure to define the APIs to manage work in the mediated core driver that is related to the physical device.

The structures in the mdev_parent_ops structure are as follows:

- dev_attr_groups: attributes of the parent device
- mdev_attr_groups: attributes of the mediated device
- supported_config: attributes to define supported configurations

The functions in the mdev_parent_ops structure are as follows:

- create: allocate basic resources in a driver for a mediated device
- remove: free resources in a driver when a mediated device is destroyed

(Note that mdev-core provides no implicit serialization of create/remove callbacks per mdev parent device, per mdev type, or any other categorization. Vendor

drivers are expected to be fully asynchronous in this respect or provide their own internal resource protection.)

The callbacks in the mdev_parent_ops structure are as follows:

- open: open callback of mediated device
- close: close callback of mediated device
- ioctl: ioctl callback of mediated device
- read : read emulation callback
- write: write emulation callback
- mmap: mmap emulation callback

A driver should use the mdev_parent_ops structure in the function call to register itself with the mdev core driver:

```
extern int  mdev_register_device(struct device *dev,
                                 const struct mdev_parent_ops *ops);
```

However, the mdev_parent_ops structure is not required in the function call that a driver should use to unregister itself with the mdev core driver:

```
extern void mdev_unregister_device(struct device *dev);
```

89.3 Mediated Device Management Interface Through sysfs

The management interface through sysfs enables user space software, such as libvirt, to query and configure mediated devices in a hardware-agnostic fashion. This management interface provides flexibility to the underlying physical device' s driver to support features such as:

- Mediated device hot plug
- Multiple mediated devices in a single virtual machine
- Multiple mediated devices from different physical devices

89.3.1 Links in the mdev_bus Class Directory

The /sys/class/mdev_bus/ directory contains links to devices that are registered with the mdev core driver.

89.3.2 Directories and files under the sysfs for Each Physical Device

```
|- [parent physical device]
|--- Vendor-specific-attributes [optional]
|--- [mdev_supported_types]
|      |--- [<type-id>]
|      |    |--- create
|      |    |--- name
|      |    |--- available_instances
|      |    |--- device_api
|      |    |--- description
|      |    |--- [devices]
|      |--- [<type-id>]
|      |    |--- create
|      |    |--- name
|      |    |--- available_instances
|      |    |--- device_api
|      |    |--- description
|      |    |--- [devices]
|      |--- [<type-id>]
|           |--- create
|           |--- name
|           |--- available_instances
|           |--- device_api
|           |--- description
|           |--- [devices]
```

- [mdev_supported_types]

  The list of currently supported mediated device types and their details.

  [<type-id>], device_api, and available_instances are mandatory attributes that should be provided by vendor driver.

- [<type-id>]

  The [<type-id>] name is created by adding the device driver string as a prefix to the string provided by the vendor driver. This format of this name is as follows:

  ```
  sprintf(buf, "%s-%s", dev_driver_string(parent->dev), group->name);
  ```

  (or using mdev_parent_dev(mdev) to arrive at the parent device outside of the core mdev code)

- device_api

  This attribute should show which device API is being created, for example, “vfio-pci” for a PCI device.

- available_instances

  This attribute should show the number of devices of type <type-id> that can be created.

- [device]

  This directory contains links to the devices of type <type-id> that have been created.

- name

  This attribute should show human readable name. This is optional attribute.

- description

  This attribute should show brief features/description of the type. This is optional attribute.

89.3.3 Directories and Files Under the sysfs for Each mdev Device

```
|- [parent phy device]
|--- [$MDEV_UUID]
       |--- remove
       |--- mdev_type {link to its type}
       |--- vendor-specific-attributes [optional]
```

- remove (write only)

Writing '1' to the 'remove' file destroys the mdev device. The vendor driver can fail the remove() callback if that device is active and the vendor driver doesn't support hot unplug.

Example:

```
# echo 1 > /sys/bus/mdev/devices/$mdev_UUID/remove
```

89.3.4 Mediated device Hot plug

Mediated devices can be created and assigned at runtime. The procedure to hot plug a mediated device is the same as the procedure to hot plug a PCI device.

89.4 Translation APIs for Mediated Devices

The following APIs are provided for translating user pfn to host pfn in a VFIO driver:

```
extern int vfio_pin_pages(struct device *dev, unsigned long *user_pfn,
                          int npage, int prot, unsigned long *phys_pfn);

extern int vfio_unpin_pages(struct device *dev, unsigned long *user_pfn,
                            int npage);
```

These functions call back into the back-end IOMMU module by using the pin_pages and unpin_pages callbacks of the struct vfio_iommu_driver_ops[4]. Currently these callbacks are supported in the TYPE1 IOMMU module. To enable them for other IOMMU backend modules, such as PPC64 sPAPR module, they need to provide these two callback functions.

89.5 Using the Sample Code

mtty.c in samples/vfio-mdev/ directory is a sample driver program to demonstrate how to use the mediated device framework.

The sample driver creates an mdev device that simulates a serial port over a PCI card.

1. Build and load the mtty.ko module.

   This step creates a dummy device, /sys/devices/virtual/mtty/mtty/

   Files in this device directory in sysfs are similar to the following:

```
# tree /sys/devices/virtual/mtty/mtty/
   /sys/devices/virtual/mtty/mtty/
   |-- mdev_supported_types
   |   |-- mtty-1
   |   |   |-- available_instances
   |   |   |-- create
   |   |   |-- device_api
   |   |   |-- devices
   |   |   `-- name
   |   `-- mtty-2
   |       |-- available_instances
   |       |-- create
   |       |-- device_api
   |       |-- devices
   |       `-- name
   |-- mtty_dev
   |   `-- sample_mtty_dev
   |-- power
   |   |-- autosuspend_delay_ms
   |   |-- control
   |   |-- runtime_active_time
   |   |-- runtime_status
   |   `-- runtime_suspended_time
   |-- subsystem -> ../../../../class/mtty
   `-- uevent
```

2. Create a mediated device by using the dummy device that you created in the previous step:

```
# echo "83b8f4f2-509f-382f-3c1e-e6bfe0fa1001" >    \
         /sys/devices/virtual/mtty/mtty/mdev_supported_types/mtty-2/
↪create
```

3. Add parameters to qemu-kvm:

```
-device vfio-pci,\
 sysfsdev=/sys/bus/mdev/devices/83b8f4f2-509f-382f-3c1e-e6bfe0fa1001
```

4. Boot the VM.

   In the Linux guest VM, with no hardware on the host, the device appears as follows:

```
# lspci -s 00:05.0 -xxvv
00:05.0 Serial controller: Device 4348:3253 (rev 10) (prog-if 02␣
↪[16550])
        Subsystem: Device 4348:3253
        Physical Slot: 5
        Control: I/O+ Mem- BusMaster- SpecCycle- MemWINV- VGASnoop-␣
↪ParErr-
Stepping- SERR- FastB2B- DisINTx-
        Status: Cap- 66MHz- UDF- FastB2B- ParErr- DEVSEL=medium >
↪TAbort-
<TAbort- <MAbort- >SERR- <PERR- INTx-
        Interrupt: pin A routed to IRQ 10
        Region 0: I/O ports at c150 [size=8]
        Region 1: I/O ports at c158 [size=8]
        Kernel driver in use: serial
00: 48 43 53 32 01 00 00 02 10 02 00 07 00 00 00 00
10: 51 c1 00 00 59 c1 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 48 43 53 32
30: 00 00 00 00 00 00 00 00 00 00 00 00 0a 01 00 00

In the Linux guest VM, dmesg output for the device is as follows:

serial 0000:00:05.0: PCI INT A -> Link[LNKA] -> GSI 10 (level, high) -
↪> IRQ 10
0000:00:05.0: ttyS1 at I/O 0xc150 (irq = 10) is a 16550A
0000:00:05.0: ttyS2 at I/O 0xc158 (irq = 10) is a 16550A
```

5. In the Linux guest VM, check the serial ports:

```
# setserial -g /dev/ttyS*
/dev/ttyS0, UART: 16550A, Port: 0x03f8, IRQ: 4
/dev/ttyS1, UART: 16550A, Port: 0xc150, IRQ: 10
/dev/ttyS2, UART: 16550A, Port: 0xc158, IRQ: 10
```

6. Using minicom or any terminal emulation program, open port /dev/ttyS1 or /dev/ttyS2 with hardware flow control disabled.

7. Type data on the minicom terminal or send data to the terminal emulation program and read the data.

   Data is loop backed from hosts mtty driver.

8. Destroy the mediated device that you created:

```
# echo 1 > /sys/bus/mdev/devices/83b8f4f2-509f-382f-3c1e-e6bfe0fa1001/
↪remove
```

89.6 References

1. See Documentation/driver-api/vfio.rst for more information on VFIO.
2. struct mdev_driver in include/linux/mdev.h
3. struct mdev_parent_ops in include/linux/mdev.h
4. struct vfio_iommu_driver_ops in include/linux/vfio.h

CHAPTER

NINETY

VFIO - “VIRTUAL FUNCTION I/O”[1]

Many modern system now provide DMA and interrupt remapping facilities to help ensure I/O devices behave within the boundaries they’ve been allotted. This includes x86 hardware with AMD-Vi and Intel VT-d, POWER systems with Partitionable Endpoints (PEs) and embedded PowerPC systems such as Freescale PAMU. The VFIO driver is an IOMMU/device agnostic framework for exposing direct device access to userspace, in a secure, IOMMU protected environment. In other words, this allows safe[2], non-privileged, userspace drivers.

Why do we want that? Virtual machines often make use of direct device access (“device assignment”) when configured for the highest possible I/O performance. From a device and host perspective, this simply turns the VM into a userspace driver, with the benefits of significantly reduced latency, higher bandwidth, and direct use of bare-metal device drivers[3].

Some applications, particularly in the high performance computing field, also benefit from low-overhead, direct device access from userspace. Examples include network adapters (often non-TCP/IP based) and compute accelerators. Prior to VFIO, these drivers had to either go through the full development cycle to become proper upstream driver, be maintained out of tree, or make use of the UIO framework, which has no notion of IOMMU protection, limited interrupt support, and requires root privileges to access things like PCI configuration space.

The VFIO driver framework intends to unify these, replacing both the KVM PCI specific device assignment code as well as provide a more secure, more featureful userspace driver environment than UIO.

[1] VFIO was originally an acronym for “Virtual Function I/O” in its initial implementation by Tom Lyon while as Cisco. We’ve since outgrown the acronym, but it’s catchy.

[2] “safe” also depends upon a device being “well behaved”. It’s possible for multi-function devices to have backdoors between functions and even for single function devices to have alternative access to things like PCI config space through MMIO registers. To guard against the former we can include additional precautions in the IOMMU driver to group multi-function PCI devices together (iommu=group_mf). The latter we can’t prevent, but the IOMMU should still provide isolation. For PCI, SR-IOV Virtual Functions are the best indicator of “well behaved”, as these are designed for virtualization usage models.

[3] As always there are trade-offs to virtual machine device assignment that are beyond the scope of VFIO. It’s expected that future IOMMU technologies will reduce some, but maybe not all, of these trade-offs.

90.1 Groups, Devices, and IOMMUs

Devices are the main target of any I/O driver. Devices typically create a programming interface made up of I/O access, interrupts, and DMA. Without going into the details of each of these, DMA is by far the most critical aspect for maintaining a secure environment as allowing a device read-write access to system memory imposes the greatest risk to the overall system integrity.

To help mitigate this risk, many modern IOMMUs now incorporate isolation properties into what was, in many cases, an interface only meant for translation (ie. solving the addressing problems of devices with limited address spaces). With this, devices can now be isolated from each other and from arbitrary memory access, thus allowing things like secure direct assignment of devices into virtual machines.

This isolation is not always at the granularity of a single device though. Even when an IOMMU is capable of this, properties of devices, interconnects, and IOMMU topologies can each reduce this isolation. For instance, an individual device may be part of a larger multi- function enclosure. While the IOMMU may be able to distinguish between devices within the enclosure, the enclosure may not require transactions between devices to reach the IOMMU. Examples of this could be anything from a multi-function PCI device with backdoors between functions to a non-PCI-ACS (Access Control Services) capable bridge allowing redirection without reaching the IOMMU. Topology can also play a factor in terms of hiding devices. A PCIe-to-PCI bridge masks the devices behind it, making transaction appear as if from the bridge itself. Obviously IOMMU design plays a major factor as well.

Therefore, while for the most part an IOMMU may have device level granularity, any system is susceptible to reduced granularity. The IOMMU API therefore supports a notion of IOMMU groups. A group is a set of devices which is isolatable from all other devices in the system. Groups are therefore the unit of ownership used by VFIO.

While the group is the minimum granularity that must be used to ensure secure user access, it's not necessarily the preferred granularity. In IOMMUs which make use of page tables, it may be possible to share a set of page tables between different groups, reducing the overhead both to the platform (reduced TLB thrashing, reduced duplicate page tables), and to the user (programming only a single set of translations). For this reason, VFIO makes use of a container class, which may hold one or more groups. A container is created by simply opening the /dev/vfio/vfio character device.

On its own, the container provides little functionality, with all but a couple version and extension query interfaces locked away. The user needs to add a group into the container for the next level of functionality. To do this, the user first needs to identify the group associated with the desired device. This can be done using the sysfs links described in the example below. By unbinding the device from the host driver and binding it to a VFIO driver, a new VFIO group will appear for the group as /dev/vfio/$GROUP, where $GROUP is the IOMMU group number of which the device is a member. If the IOMMU group contains multiple devices, each will need to be bound to a VFIO driver before operations on the VFIO group are allowed (it's also sufficient to only unbind the device from host drivers if a VFIO driver is unavailable; this will make the group available, but not that particular device).

TBD - interface for disabling driver probing/locking a device.

Once the group is ready, it may be added to the container by opening the VFIO group character device (/dev/vfio/$GROUP) and using the VFIO_GROUP_SET_CONTAINER ioctl, passing the file descriptor of the previously opened container file. If desired and if the IOMMU driver supports sharing the IOMMU context between groups, multiple groups may be set to the same container. If a group fails to set to a container with existing groups, a new empty container will need to be used instead.

With a group (or groups) attached to a container, the remaining ioctls become available, enabling access to the VFIO IOMMU interfaces. Additionally, it now becomes possible to get file descriptors for each device within a group using an ioctl on the VFIO group file descriptor.

The VFIO device API includes ioctls for describing the device, the I/O regions and their read/write/mmap offsets on the device descriptor, as well as mechanisms for describing and registering interrupt notifications.

90.2 VFIO Usage Example

Assume user wants to access PCI device 0000:06:0d.0:

```
$ readlink /sys/bus/pci/devices/0000:06:0d.0/iommu_group
../../../../kernel/iommu_groups/26
```

This device is therefore in IOMMU group 26. This device is on the pci bus, therefore the user will make use of vfio-pci to manage the group:

```
# modprobe vfio-pci
```

Binding this device to the vfio-pci driver creates the VFIO group character devices for this group:

```
$ lspci -n -s 0000:06:0d.0
06:0d.0 0401: 1102:0002 (rev 08)
# echo 0000:06:0d.0 > /sys/bus/pci/devices/0000:06:0d.0/driver/unbind
# echo 1102 0002 > /sys/bus/pci/drivers/vfio-pci/new_id
```

Now we need to look at what other devices are in the group to free it for use by VFIO:

```
$ ls -l /sys/bus/pci/devices/0000:06:0d.0/iommu_group/devices
total 0
lrwxrwxrwx. 1 root root 0 Apr 23 16:13 0000:00:1e.0 ->
        ../../../../devices/pci0000:00/0000:00:1e.0
lrwxrwxrwx. 1 root root 0 Apr 23 16:13 0000:06:0d.0 ->
        ../../../../devices/pci0000:00/0000:00:1e.0/0000:06:0d.0
lrwxrwxrwx. 1 root root 0 Apr 23 16:13 0000:06:0d.1 ->
        ../../../../devices/pci0000:00/0000:00:1e.0/0000:06:0d.1
```

This device is behind a PCIe-to-PCI bridge[4], therefore we also need to add de-

[4] In this case the device is below a PCI bridge, so transactions from either function of the device are indistinguishable to the iommu:

vice 0000:06:0d.1 to the group following the same procedure as above. Device 0000:00:1e.0 is a bridge that does not currently have a host driver, therefore it's not required to bind this device to the vfio-pci driver (vfio-pci does not currently support PCI bridges).

The final step is to provide the user with access to the group if unprivileged operation is desired (note that /dev/vfio/vfio provides no capabilities on its own and is therefore expected to be set to mode 0666 by the system):

```
# chown user:user /dev/vfio/26
```

The user now has full access to all the devices and the iommu for this group and can access them as follows:

```
int container, group, device, i;
struct vfio_group_status group_status =
                                 { .argsz = sizeof(group_status) };
struct vfio_iommu_type1_info iommu_info = { .argsz = sizeof(iommu_info) };
struct vfio_iommu_type1_dma_map dma_map = { .argsz = sizeof(dma_map) };
struct vfio_device_info device_info = { .argsz = sizeof(device_info) };

/* Create a new container */
container = open("/dev/vfio/vfio", O_RDWR);

if (ioctl(container, VFIO_GET_API_VERSION) != VFIO_API_VERSION)
        /* Unknown API version */

if (!ioctl(container, VFIO_CHECK_EXTENSION, VFIO_TYPE1_IOMMU))
        /* Doesn't support the IOMMU driver we want. */

/* Open the group */
group = open("/dev/vfio/26", O_RDWR);

/* Test the group is viable and available */
ioctl(group, VFIO_GROUP_GET_STATUS, &group_status);

if (!(group_status.flags & VFIO_GROUP_FLAGS_VIABLE))
        /* Group is not viable (ie, not all devices bound for vfio) */

/* Add the group to the container */
ioctl(group, VFIO_GROUP_SET_CONTAINER, &container);

/* Enable the IOMMU model we want */
ioctl(container, VFIO_SET_IOMMU, VFIO_TYPE1_IOMMU);

/* Get addition IOMMU info */
ioctl(container, VFIO_IOMMU_GET_INFO, &iommu_info);

/* Allocate some space and setup a DMA mapping */
dma_map.vaddr = mmap(0, 1024 * 1024, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
```

(continues on next page)

```
-[0000:00]-+-1e.0-[06]--+-0d.0
                        \-0d.1

00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev 90)
```

(continued from previous page)

```
dma_map.size = 1024 * 1024;
dma_map.iova = 0; /* 1MB starting at 0x0 from device view */
dma_map.flags = VFIO_DMA_MAP_FLAG_READ | VFIO_DMA_MAP_FLAG_WRITE;

ioctl(container, VFIO_IOMMU_MAP_DMA, &dma_map);

/* Get a file descriptor for the device */
device = ioctl(group, VFIO_GROUP_GET_DEVICE_FD, "0000:06:0d.0");

/* Test and setup the device */
ioctl(device, VFIO_DEVICE_GET_INFO, &device_info);

for (i = 0; i < device_info.num_regions; i++) {
        struct vfio_region_info reg = { .argsz = sizeof(reg) };

        reg.index = i;

        ioctl(device, VFIO_DEVICE_GET_REGION_INFO, &reg);

        /* Setup mappings... read/write offsets, mmaps
         * For PCI devices, config space is a region */
}

for (i = 0; i < device_info.num_irqs; i++) {
        struct vfio_irq_info irq = { .argsz = sizeof(irq) };

        irq.index = i;

        ioctl(device, VFIO_DEVICE_GET_IRQ_INFO, &irq);

        /* Setup IRQs... eventfds, VFIO_DEVICE_SET_IRQS */
}

/* Gratuitous device reset and go... */
ioctl(device, VFIO_DEVICE_RESET);
```

90.3 VFIO User API

Please see include/linux/vfio.h for complete API documentation.

90.4 VFIO bus driver API

VFIO bus drivers, such as vfio-pci make use of only a few interfaces into VFIO core. When devices are bound and unbound to the driver, the driver should call vfio_add_group_dev() and vfio_del_group_dev() respectively:

```
extern int vfio_add_group_dev(struct device *dev,
                              const struct vfio_device_ops *ops,
                              void *device_data);

extern void *vfio_del_group_dev(struct device *dev);
```

vfio_add_group_dev() indicates to the core to begin tracking the iommu_group of the specified dev and register the dev as owned by a VFIO bus driver. The driver provides an ops structure for callbacks similar to a file operations structure:

```
struct vfio_device_ops {
        int     (*open)(void *device_data);
        void    (*release)(void *device_data);
        ssize_t (*read)(void *device_data, char __user *buf,
                        size_t count, loff_t *ppos);
        ssize_t (*write)(void *device_data, const char __user *buf,
                         size_t size, loff_t *ppos);
        long    (*ioctl)(void *device_data, unsigned int cmd,
                         unsigned long arg);
        int     (*mmap)(void *device_data, struct vm_area_struct *vma);
};
```

Each function is passed the device_data that was originally registered in the vfio_add_group_dev() call above. This allows the bus driver an easy place to store its opaque, private data. The open/release callbacks are issued when a new file descriptor is created for a device (via VFIO_GROUP_GET_DEVICE_FD). The ioctl interface provides a direct pass through for VFIO_DEVICE_* ioctls. The read/write/mmap interfaces implement the device region access defined by the device's own VFIO_DEVICE_GET_REGION_INFO ioctl.

90.5 PPC64 sPAPR implementation note

This implementation has some specifics:

1) On older systems (POWER7 with P5IOC2/IODA1) only one IOMMU group per container is supported as an IOMMU table is allocated at the boot time, one table per a IOMMU group which is a Partitionable Endpoint (PE) (PE is often a PCI domain but not always).

   Newer systems (POWER8 with IODA2) have improved hardware design which allows to remove this limitation and have multiple IOMMU groups per a VFIO container.

2) The hardware supports so called DMA windows - the PCI address range within which DMA transfer is allowed, any attempt to access address space out of the window leads to the whole PE isolation.

3) PPC64 guests are paravirtualized but not fully emulated. There is an API to map/unmap pages for DMA, and it normally maps 1..32 pages per call and currently there is no way to reduce the number of calls. In order to make things faster, the map/unmap handling has been implemented in real mode which provides an excellent performance which has limitations such as inability to do locked pages accounting in real time.

4) According to sPAPR specification, A Partitionable Endpoint (PE) is an I/O subtree that can be treated as a unit for the purposes of partitioning and error recovery. A PE may be a single or multi-function IOA (IO Adapter), a function of a multi-function IOA, or multiple IOAs (possibly including switch and bridge structures above the multiple IOAs). PPC64 guests detect PCI errors

and recover from them via EEH RTAS services, which works on the basis of additional ioctl commands.

So 4 additional ioctls have been added:

VFIO_IOMMU_SPAPR_TCE_GET_INFO returns the size and the start of the DMA window on the PCI bus.

VFIO_IOMMU_ENABLE enables the container. The locked pages accounting is done at this point. This lets user first to know what the DMA window is and adjust rlimit before doing any real job.

VFIO_IOMMU_DISABLE disables the container.

VFIO_EEH_PE_OP provides an API for EEH setup, error detection and recovery.

The code flow from the example above should be slightly changed:

```
struct vfio_eeh_pe_op pe_op = { .argsz = sizeof(pe_op), .flags = 0 };

.....
/* Add the group to the container */
ioctl(group, VFIO_GROUP_SET_CONTAINER, &container);

/* Enable the IOMMU model we want */
ioctl(container, VFIO_SET_IOMMU, VFIO_SPAPR_TCE_IOMMU)

/* Get addition sPAPR IOMMU info */
vfio_iommu_spapr_tce_info spapr_iommu_info;
ioctl(container, VFIO_IOMMU_SPAPR_TCE_GET_INFO, &spapr_iommu_info);

if (ioctl(container, VFIO_IOMMU_ENABLE))
        /* Cannot enable container, may be low rlimit */

/* Allocate some space and setup a DMA mapping */
dma_map.vaddr = mmap(0, 1024 * 1024, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);

dma_map.size = 1024 * 1024;
dma_map.iova = 0; /* 1MB starting at 0x0 from device view */
dma_map.flags = VFIO_DMA_MAP_FLAG_READ | VFIO_DMA_MAP_FLAG_WRITE;

/* Check here is .iova/.size are within DMA window from spapr_iommu_
↪info */
ioctl(container, VFIO_IOMMU_MAP_DMA, &dma_map);

/* Get a file descriptor for the device */
device = ioctl(group, VFIO_GROUP_GET_DEVICE_FD, "0000:06:0d.0");

....

/* Gratuitous device reset and go... */
ioctl(device, VFIO_DEVICE_RESET);

/* Make sure EEH is supported */
ioctl(container, VFIO_CHECK_EXTENSION, VFIO_EEH);
```

(continues on next page)

(continued from previous page)

```
/* Enable the EEH functionality on the device */
pe_op.op = VFIO_EEH_PE_ENABLE;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* You're suggested to create additional data struct to represent
 * PE, and put child devices belonging to same IOMMU group to the
 * PE instance for later reference.
 */

/* Check the PE's state and make sure it's in functional state */
pe_op.op = VFIO_EEH_PE_GET_STATE;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* Save device state using pci_save_state().
 * EEH should be enabled on the specified device.
 */

....

/* Inject EEH error, which is expected to be caused by 32-bits
 * config load.
 */
pe_op.op = VFIO_EEH_PE_INJECT_ERR;
pe_op.err.type = EEH_ERR_TYPE_32;
pe_op.err.func = EEH_ERR_FUNC_LD_CFG_ADDR;
pe_op.err.addr = 0ul;
pe_op.err.mask = 0ul;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

....

/* When 0xFF's returned from reading PCI config space or IO BARs
 * of the PCI device. Check the PE's state to see if that has been
 * frozen.
 */
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* Waiting for pending PCI transactions to be completed and don't
 * produce any more PCI traffic from/to the affected PE until
 * recovery is finished.
 */

/* Enable IO for the affected PE and collect logs. Usually, the
 * standard part of PCI config space, AER registers are dumped
 * as logs for further analysis.
 */
pe_op.op = VFIO_EEH_PE_UNFREEZE_IO;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/*
 * Issue PE reset: hot or fundamental reset. Usually, hot reset
 * is enough. However, the firmware of some PCI adapters would
 * require fundamental reset.
 */
pe_op.op = VFIO_EEH_PE_RESET_HOT;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);
```

(continues on next page)

(continued from previous page)

```
pe_op.op = VFIO_EEH_PE_RESET_DEACTIVATE;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* Configure the PCI bridges for the affected PE */
pe_op.op = VFIO_EEH_PE_CONFIGURE;
ioctl(container, VFIO_EEH_PE_OP, &pe_op);

/* Restored state we saved at initialization time. pci_restore_state()
 * is good enough as an example.
 */

/* Hopefully, error is recovered successfully. Now, you can resume to
 * start PCI traffic to/from the affected PE.
 */

....
```

5) There is v2 of SPAPR TCE IOMMU. It deprecates VFIO_IOMMU_ENABLE/ VFIO_IOMMU_DISABLE and implements 2 new ioctls: VFIO_IOMMU_SPAPR_REGISTER_MEMORY and VFIO_IOMMU_SPAPR_UNREGISTER_MEMORY (which are unsupported in v1 IOMMU).

   PPC64 paravirtualized guests generate a lot of map/unmap requests, and the handling of those includes pinning/unpinning pages and updating mm::locked_vm counter to make sure we do not exceed the rlimit. The v2 IOMMU splits accounting and pinning into separate operations:

   - VFIO_IOMMU_SPAPR_REGISTER_MEMORY/VFIO_IOMMU_SPAPR_UNREGISTER_M ioctls receive a user space address and size of the block to be pinned. Bisecting is not supported and VFIO_IOMMU_UNREGISTER_MEMORY is expected to be called with the exact address and size used for registering the memory block. The userspace is not expected to call these often. The ranges are stored in a linked list in a VFIO container.
   - VFIO_IOMMU_MAP_DMA/VFIO_IOMMU_UNMAP_DMA ioctls only update the actual IOMMU table and do not do pinning; instead these check that the userspace address is from pre-registered range.

   This separation helps in optimizing DMA for guests.

6) sPAPR specification allows guests to have an additional DMA window(s) on a PCI bus with a variable page size. Two ioctls have been added to support this: VFIO_IOMMU_SPAPR_TCE_CREATE and VFIO_IOMMU_SPAPR_TCE_REMOVE. The platform has to support the functionality or error will be returned to the userspace. The existing hardware supports up to 2 DMA windows, one is 2GB long, uses 4K pages and called “default 32bit window” ; the other can be as big as entire RAM, use different page size, it is optional - guests create those in run-time if the guest driver supports 64bit DMA.

   VFIO_IOMMU_SPAPR_TCE_CREATE receives a page shift, a DMA window size and a number of TCE table levels (if a TCE table is going to be big enough and the kernel may not be able to allocate enough of physically contiguous memory). It creates a new window in the available slot and returns the bus

address where the new window starts. Due to hardware limitation, the user space cannot choose the location of DMA windows.

VFIO_IOMMU_SPAPR_TCE_REMOVE receives the bus start address of the window and removes it.

CHAPTER

NINETYONE

XILINX FPGA

91.1 Xilinx Zynq MPSoC EEMI Documentation

91.1.1 Xilinx Zynq MPSoC Firmware Interface

The zynqmp-firmware node describes the interface to platform firmware. ZynqMP has an interface to communicate with secure firmware. Firmware driver provides an interface to firmware APIs. Interface APIs can be used by any driver to communicate with PMC(Platform Management Controller).

91.1.2 Embedded Energy Management Interface (EEMI)

The embedded energy management interface is used to allow software components running across different processing clusters on a chip or device to communicate with a power management controller (PMC) on a device to issue or respond to power management requests.

EEMI ops is a structure containing all eemi APIs supported by Zynq MPSoC. The zynqmp-firmware driver maintain all EEMI APIs in zynqmp_eemi_ops structure. Any driver who want to communicate with PMC using EEMI APIs can call zynqmp_pm_get_eemi_ops().

Example of EEMI ops:

```
/* zynqmp-firmware driver maintain all EEMI APIs */
struct zynqmp_eemi_ops {
        int (*get_api_version)(u32 *version);
        int (*query_data)(struct zynqmp_pm_query_data qdata, u32 *out);
};

static const struct zynqmp_eemi_ops eemi_ops = {
        .get_api_version = zynqmp_pm_get_api_version,
        .query_data = zynqmp_pm_query_data,
};
```

Example of EEMI ops usage:

```
static const struct zynqmp_eemi_ops *eemi_ops;
u32 ret_payload[PAYLOAD_ARG_CNT];
int ret;
```

(continues on next page)

(continued from previous page)

```
eemi_ops = zynqmp_pm_get_eemi_ops();
if (IS_ERR(eemi_ops))
        return PTR_ERR(eemi_ops);

ret = eemi_ops->query_data(qdata, ret_payload);
```

91.1.3 IOCTL

IOCTL API is for device control and configuration. It is not a system IOCTL but it is an EEMI API. This API can be used by master to control any device specific configuration. IOCTL definitions can be platform specific. This API also manage shared device configuration.

The following IOCTL IDs are valid for device control: - IOCTL_SET_PLL_FRAC_MODE 8 - IOCTL_GET_PLL_FRAC_MODE 9 - IOCTL_SET_PLL_FRAC_DATA 10 - IOCTL_GET_PLL_FRAC_DATA 11

Refer EEMI API guide [0] for IOCTL specific parameters and other EEMI APIs.

91.1.4 References

[0] Embedded Energy Management Interface (EEMI) API guide: https://www.xilinx.com/support/documentation/user_guides/ug1200-eemi-api.pdf

CHAPTER

NINETYTWO

XILLYBUS DRIVER FOR GENERIC FPGA INTERFACE

Author Eli Billauer, Xillybus Ltd. (http://xillybus.com)

Email eli.billauer@gmail.com or as advertised on Xillybus' site.

92.1 Introduction

92.1.1 Background

An FPGA (Field Programmable Gate Array) is a piece of logic hardware, which can be programmed to become virtually anything that is usually found as a dedicated chipset: For instance, a display adapter, network interface card, or even a processor with its peripherals. FPGAs are the LEGO of hardware: Based upon certain building blocks, you make your own toys the way you like them. It's usually pointless to reimplement something that is already available on the market as a chipset, so FPGAs are mostly used when some special functionality is needed, and the production volume is relatively low (hence not justifying the development of an ASIC).

The challenge with FPGAs is that everything is implemented at a very low level, even lower than assembly language. In order to allow FPGA designers to focus on their specific project, and not reinvent the wheel over and over again, pre-designed building blocks, IP cores, are often used. These are the FPGA parallels of library functions. IP cores may implement certain mathematical functions, a functional unit (e.g. a USB interface), an entire processor (e.g. ARM) or anything that might come handy. Think of them as a building block, with electrical wires dangling on the sides for connection to other blocks.

One of the daunting tasks in FPGA design is communicating with a fullblown operating system (actually, with the processor running it): Implementing the low-level bus protocol and the somewhat higher-level interface with the host (registers, interrupts, DMA etc.) is a project in itself. When the FPGA's function is a well-known one (e.g. a video adapter card, or a NIC), it can make sense to design the FPGA's interface logic specifically for the project. A special driver is then written to present the FPGA as a well-known interface to the kernel and/or user space. In that case, there is no reason to treat the FPGA differently than any device on the bus.

It's however common that the desired data communication doesn't fit any well-known peripheral function. Also, the effort of designing an elegant abstraction for the data exchange is often considered too big. In those cases, a quicker and

possibly less elegant solution is sought: The driver is effectively written as a user space program, leaving the kernel space part with just elementary data transport. This still requires designing some interface logic for the FPGA, and write a simple ad-hoc driver for the kernel.

92.1.2 Xillybus Overview

Xillybus is an IP core and a Linux driver. Together, they form a kit for elementary data transport between an FPGA and the host, providing pipe-like data streams with a straightforward user interface. It's intended as a low- effort solution for mixed FPGA-host projects, for which it makes sense to have the project-specific part of the driver running in a user-space program.

Since the communication requirements may vary significantly from one FPGA project to another (the number of data pipes needed in each direction and their attributes), there isn't one specific chunk of logic being the Xillybus IP core. Rather, the IP core is configured and built based upon a specification given by its end user.

Xillybus presents independent data streams, which resemble pipes or TCP/IP communication to the user. At the host side, a character device file is used just like any pipe file. On the FPGA side, hardware FIFOs are used to stream the data. This is contrary to a common method of communicating through fixed- sized buffers (even though such buffers are used by Xillybus under the hood). There may be more than a hundred of these streams on a single IP core, but also no more than one, depending on the configuration.

In order to ease the deployment of the Xillybus IP core, it contains a simple data structure which completely defines the core's configuration. The Linux driver fetches this data structure during its initialization process, and sets up the DMA buffers and character devices accordingly. As a result, a single driver is used to work out of the box with any Xillybus IP core.

The data structure just mentioned should not be confused with PCI's configuration space or the Flattened Device Tree.

92.2 Usage

92.2.1 User interface

On the host, all interface with Xillybus is done through /dev/xillybus_* device files, which are generated automatically as the drivers loads. The names of these files depend on the IP core that is loaded in the FPGA (see Probing below). To communicate with the FPGA, open the device file that corresponds to the hardware FIFO you want to send data or receive data from, and use plain write() or read() calls, just like with a regular pipe. In particular, it makes perfect sense to go:

```
$ cat mydata > /dev/xillybus_thisfifo

$ cat /dev/xillybus_thatfifo > hisdata
```

possibly pressing CTRL-C as some stage, even though the xillybus_* pipes have the capability to send an EOF (but may not use it).

The driver and hardware are designed to behave sensibly as pipes, including:

- Supporting non-blocking I/O (by setting O_NONBLOCK on open()).
- Supporting poll() and select().
- Being bandwidth efficient under load (using DMA) but also handle small pieces of data sent across (like TCP/IP) by autoflushing.

A device file can be read only, write only or bidirectional. Bidirectional device files are treated like two independent pipes (except for sharing a “channel” structure in the implementation code).

92.2.2 Synchronization

Xillybus pipes are configured (on the IP core) to be either synchronous or asynchronous. For a synchronous pipe, write() returns successfully only after some data has been submitted and acknowledged by the FPGA. This slows down bulk data transfers, and is nearly impossible for use with streams that require data at a constant rate: There is no data transmitted to the FPGA between write() calls, in particular when the process loses the CPU.

When a pipe is configured asynchronous, write() returns if there was enough room in the buffers to store any of the data in the buffers.

For FPGA to host pipes, asynchronous pipes allow data transfer from the FPGA as soon as the respective device file is opened, regardless of if the data has been requested by a read() call. On synchronous pipes, only the amount of data requested by a read() call is transmitted.

In summary, for synchronous pipes, data between the host and FPGA is transmitted only to satisfy the read() or write() call currently handled by the driver, and those calls wait for the transmission to complete before returning.

Note that the synchronization attribute has nothing to do with the possibility that read() or write() completes less bytes than requested. There is a separate configuration flag (“allowpartial”) that determines whether such a partial completion is allowed.

92.2.3 Seekable pipes

A synchronous pipe can be configured to have the stream’s position exposed to the user logic at the FPGA. Such a pipe is also seekable on the host API. With this feature, a memory or register interface can be attached on the FPGA side to the seekable stream. Reading or writing to a certain address in the attached memory is done by seeking to the desired address, and calling read() or write() as required.

92.3 Internals

92.3.1 Source code organization

The Xillybus driver consists of a core module, xillybus_core.c, and modules that depend on the specific bus interface (xillybus_of.c and xillybus_pcie.c).

The bus specific modules are those probed when a suitable device is found by the kernel. Since the DMA mapping and synchronization functions, which are bus dependent by their nature, are used by the core module, a xilly_endpoint_hardware structure is passed to the core module on initialization. This structure is populated with pointers to wrapper functions which execute the DMA-related operations on the bus.

92.3.2 Pipe attributes

Each pipe has a number of attributes which are set when the FPGA component (IP core) is built. They are fetched from the IDT (the data structure which defines the core's configuration, see Probing below) by xilly_setupchannels() in xillybus_core.c as follows:

- is_writebuf: The pipe's direction. A non-zero value means it's an FPGA to host pipe (the FPGA "writes").
- channelnum: The pipe's identification number in communication between the host and FPGA.
- format: The underlying data width. See Data Granularity below.
- allowpartial: A non-zero value means that a read() or write() (whichever applies) may return with less than the requested number of bytes. The common choice is a non-zero value, to match standard UNIX behavior.
- synchronous: A non-zero value means that the pipe is synchronous. See Synchronization above.
- bufsize: Each DMA buffer's size. Always a power of two.
- bufnum: The number of buffers allocated for this pipe. Always a power of two.
- exclusive_open: A non-zero value forces exclusive opening of the associated device file. If the device file is bidirectional, and already opened only in one direction, the opposite direction may be opened once.
- seekable: A non-zero value indicates that the pipe is seekable. See Seekable pipes above.
- supports_nonempty: A non-zero value (which is typical) indicates that the hardware will send the messages that are necessary to support select() and poll() for this pipe.

92.3.3 Host never reads from the FPGA

Even though PCI Express is hotpluggable in general, a typical motherboard doesn't expect a card to go away all of the sudden. But since the PCIe card is based upon reprogrammable logic, a sudden disappearance from the bus is quite likely as a result of an accidental reprogramming of the FPGA while the host is up. In practice, nothing happens immediately in such a situation. But if the host attempts to read from an address that is mapped to the PCI Express device, that leads to an immediate freeze of the system on some motherboards, even though the PCIe standard requires a graceful recovery.

In order to avoid these freezes, the Xillybus driver refrains completely from reading from the device's register space. All communication from the FPGA to the host is done through DMA. In particular, the Interrupt Service Routine doesn't follow the common practice of checking a status register when it's invoked. Rather, the FPGA prepares a small buffer which contains short messages, which inform the host what the interrupt was about.

This mechanism is used on non-PCIe buses as well for the sake of uniformity.

92.3.4 Channels, pipes, and the message channel

Each of the (possibly bidirectional) pipes presented to the user is allocated a data channel between the FPGA and the host. The distinction between channels and pipes is necessary only because of channel 0, which is used for interrupt- related messages from the FPGA, and has no pipe attached to it.

92.3.5 Data streaming

Even though a non-segmented data stream is presented to the user at both sides, the implementation relies on a set of DMA buffers which is allocated for each channel. For the sake of illustration, let's take the FPGA to host direction: As data streams into the respective channel's interface in the FPGA, the Xillybus IP core writes it to one of the DMA buffers. When the buffer is full, the FPGA informs the host about that (appending a XILLYMSG_OPCODE_RELEASEBUF message channel 0 and sending an interrupt if necessary). The host responds by making the data available for reading through the character device. When all data has been read, the host writes on the the FPGA's buffer control register, allowing the buffer's overwriting. Flow control mechanisms exist on both sides to prevent underflows and overflows.

This is not good enough for creating a TCP/IP-like stream: If the data flow stops momentarily before a DMA buffer is filled, the intuitive expectation is that the partial data in buffer will arrive anyhow, despite the buffer not being completed. This is implemented by adding a field in the XILLYMSG_OPCODE_RELEASEBUF message, through which the FPGA informs not just which buffer is submitted, but how much data it contains.

But the FPGA will submit a partially filled buffer only if directed to do so by the host. This situation occurs when the read() method has been blocking for XILLY_RX_TIMEOUT jiffies (currently 10 ms), after which the host commands the FPGA to submit a DMA buffer as soon as it can. This timeout mechanism balances

between bus bandwidth efficiency (preventing a lot of partially filled buffers being sent) and a latency held fairly low for tails of data.

A similar setting is used in the host to FPGA direction. The handling of partial DMA buffers is somewhat different, though. The user can tell the driver to submit all data it has in the buffers to the FPGA, by issuing a write() with the byte count set to zero. This is similar to a flush request, but it doesn't block. There is also an autoflushing mechanism, which triggers an equivalent flush roughly XILLY_RX_TIMEOUT jiffies after the last write(). This allows the user to be oblivious about the underlying buffering mechanism and yet enjoy a stream-like interface.

Note that the issue of partial buffer flushing is irrelevant for pipes having the "synchronous" attribute nonzero, since synchronous pipes don't allow data to lay around in the DMA buffers between read() and write() anyhow.

92.3.6 Data granularity

The data arrives or is sent at the FPGA as 8, 16 or 32 bit wide words, as configured by the "format"attribute. Whenever possible, the driver attempts to hide this when the pipe is accessed differently from its natural alignment. For example, reading single bytes from a pipe with 32 bit granularity works with no issues. Writing single bytes to pipes with 16 or 32 bit granularity will also work, but the driver can't send partially completed words to the FPGA, so the transmission of up to one word may be held until it's fully occupied with user data.

This somewhat complicates the handling of host to FPGA streams, because when a buffer is flushed, it may contain up to 3 bytes don't form a word in the FPGA, and hence can't be sent. To prevent loss of data, these leftover bytes need to be moved to the next buffer. The parts in xillybus_core.c that mention "leftovers" in some way are related to this complication.

92.3.7 Probing

As mentioned earlier, the number of pipes that are created when the driver loads and their attributes depend on the Xillybus IP core in the FPGA. During the driver's initialization, a blob containing configuration info, the Interface Description Table (IDT), is sent from the FPGA to the host. The bootstrap process is done in three phases:

1. Acquire the length of the IDT, so a buffer can be allocated for it. This is done by sending a quiesce command to the device, since the acknowledge for this command contains the IDT's buffer length.
2. Acquire the IDT itself.
3. Create the interfaces according to the IDT.

92.3.8 Buffer allocation

In order to simplify the logic that prevents illegal boundary crossings of PCIe packets, the following rule applies: If a buffer is smaller than 4kB, it must not cross a 4kB boundary. Otherwise, it must be 4kB aligned. The xilly_setupchannels() functions allocates these buffers by requesting whole pages from the kernel, and diving them into DMA buffers as necessary. Since all buffers' sizes are powers of two, it's possible to pack any set of such buffers, with a maximal waste of one page of memory.

All buffers are allocated when the driver is loaded. This is necessary, since large continuous physical memory segments are sometimes requested, which are more likely to be available when the system is freshly booted.

The allocation of buffer memory takes place in the same order they appear in the IDT. The driver relies on a rule that the pipes are sorted with decreasing buffer size in the IDT. If a requested buffer is larger or equal to a page, the necessary number of pages is requested from the kernel, and these are used for this buffer. If the requested buffer is smaller than a page, one single page is requested from the kernel, and that page is partially used. Or, if there already is a partially used page at hand, the buffer is packed into that page. It can be shown that all pages requested from the kernel (except possibly for the last) are 100% utilized this way.

92.3.9 The "nonempty" message (supporting poll)

In order to support the "poll" method (and hence select()), there is a small catch regarding the FPGA to host direction: The FPGA may have filled a DMA buffer with some data, but not submitted that buffer. If the host waited for the buffer's submission by the FPGA, there would be a possibility that the FPGA side has sent data, but a select() call would still block, because the host has not received any notification about this. This is solved with XILLYMSG_OPCODE_NONEMPTY messages sent by the FPGA when a channel goes from completely empty to containing some data.

These messages are used only to support poll() and select(). The IP core can be configured not to send them for a slight reduction of bandwidth.

CHAPTER

NINETYTHREE

WRITING DEVICE DRIVERS FOR ZORRO DEVICES

Author Written by Geert Uytterhoeven <geert@linux-m68k.org>

Last revised September 5, 2003

93.1 Introduction

The Zorro bus is the bus used in the Amiga family of computers. Thanks to Auto-Config(tm), it's 100% Plug-and-Play.

There are two types of Zorro buses, Zorro II and Zorro III:

- The Zorro II address space is 24-bit and lies within the first 16 MB of the Amiga's address map.
- Zorro III is a 32-bit extension of Zorro II, which is backwards compatible with Zorro II. The Zorro III address space lies outside the first 16 MB.

93.2 Probing for Zorro Devices

Zorro devices are found by calling `zorro_find_device()`, which returns a pointer to the `next` Zorro device with the specified Zorro ID. A probe loop for the board with Zorro ID `ZORRO_PROD_xxx` looks like:

```
struct zorro_dev *z = NULL;

while ((z = zorro_find_device(ZORRO_PROD_xxx, z))) {
    if (!zorro_request_region(z->resource.start+MY_START, MY_SIZE,
                              "My explanation"))
    ...
}
```

`ZORRO_WILDCARD` acts as a wildcard and finds any Zorro device. If your driver supports different types of boards, you can use a construct like:

```
struct zorro_dev *z = NULL;

while ((z = zorro_find_device(ZORRO_WILDCARD, z))) {
    if (z->id != ZORRO_PROD_xxx1 && z->id != ZORRO_PROD_xxx2 && ...)
        continue;
    if (!zorro_request_region(z->resource.start+MY_START, MY_SIZE,
```

(continues on next page)

(continued from previous page)

```
                                "My explanation"))
    ...
}
```

93.3 Zorro Resources

Before you can access a Zorro device's registers, you have to make sure it's not yet in use. This is done using the I/O memory space resource management functions:

```
request_mem_region()
release_mem_region()
```

Shortcuts to claim the whole device's address space are provided as well:

```
zorro_request_device
zorro_release_device
```

93.4 Accessing the Zorro Address Space

The address regions in the Zorro device resources are Zorro bus address regions. Due to the identity bus-physical address mapping on the Zorro bus, they are CPU physical addresses as well.

The treatment of these regions depends on the type of Zorro space:

- Zorro II address space is always mapped and does not have to be mapped explicitly using z_ioremap().

  Conversion from bus/physical Zorro II addresses to kernel virtual addresses and vice versa is done using:

  ```
  virt_addr = ZTWO_VADDR(bus_addr);
  bus_addr = ZTWO_PADDR(virt_addr);
  ```

- Zorro III address space must be mapped explicitly using z_ioremap() first before it can be accessed:

  ```
  virt_addr = z_ioremap(bus_addr, size);
  ...
  z_iounmap(virt_addr);
  ```

93.5 References

1. linux/include/linux/zorro.h
2. linux/include/uapi/linux/zorro.h
3. linux/include/uapi/linux/zorro_ids.h
4. linux/arch/m68k/include/asm/zorro.h
5. linux/drivers/zorro
6. /proc/bus/zorro