Linux Pci Documentation

The kernel development community

Jul 14, 2020

CONTENTS

CHAPTER

ONE

HOW TO WRITE LINUX PCI DRIVERS

Authors

- Martin Mares <mj@ucw.cz>
- Grant Grundler <grundler@parisc-linux.org>

The world of PCI is vast and full of (mostly unpleasant) surprises. Since each CPU architecture implements different chip-sets and PCI devices have different requirements (erm, "features"), the result is the PCI support in the Linux kernel is not as trivial as one would wish. This short paper tries to introduce all potential driver authors to Linux APIs for PCI device drivers.

A more complete resource is the third edition of "Linux Device Drivers" by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. LDD3 is available for free (under Creative Commons License) from: http://lwn.net/Kernel/LDD3/.

However, keep in mind that all documents are subject to "bit rot". Refer to the source code if things are not working as described here.

Please send questions/comments/patches about Linux PCI API to the "Linux PCI" <linux-pci@atrey.karlin.mff.cuni.cz> mailing list.

1.1 Structure of PCI drivers

PCI drivers "discover" PCI devices in a system via pci_register_driver(). Actually, it's the other way around. When the PCI generic code discovers a new device, the driver with a matching "description" will be notified. Details on this below.

pci_register_driver() leaves most of the probing for devices to the PCI layer and supports online insertion/removal of devices [thus supporting hot-pluggable PCI, CardBus, and Express-Card in a single driver]. pci_register_driver() call requires passing in a table of function pointers and thus dictates the high level structure of a driver.

Once the driver knows about a PCI device and takes ownership, the driver generally needs to perform the following initialization:

- Enable the device
- Request MMIO/IOP resources
- Set the DMA mask size (for both coherent and streaming DMA)
- Allocate and initialize shared control data (pci_allocate_coherent())

- Access device configuration space (if needed)
- Register IRQ handler (request_irq())
- Initialize non-PCI (i.e. LAN/SCSI/etc parts of the chip)
- Enable DMA/processing engines

When done using the device, and perhaps the module needs to be unloaded, the driver needs to take the follow steps:

- Disable the device from generating IRQs
- Release the IRQ (free_irq())
- Stop all DMA activity
- Release DMA buffers (both streaming and coherent)
- Unregister from other subsystems (e.g. scsi or netdev)
- Release MMIO/IOP resources
- Disable the device

Most of these topics are covered in the following sections. For the rest look at LDD3 or <linux/pci.h> .

If the PCI subsystem is not configured (CONFIG_PCI is not set), most of the PCI functions described below are defined as inline functions either completely empty or just returning an appropriate error codes to avoid lots of ifdefs in the drivers.

1.2 pci_register_driver() call

PCI device drivers call `pci_register_driver()` during their initialization with a pointer to a structure describing the driver (`struct pci_driver`):

struct **pci_driver**
PCI driver structure

Definition

```
struct pci_driver {
  struct list_head        node;
  const char              *name;
  const struct pci_device_id *id_table;
  int (*probe)(struct pci_dev *dev, const struct pci_device_id *id);
  void (*remove)(struct pci_dev *dev);
  int (*suspend)(struct pci_dev *dev, pm_message_t state);
  int (*resume)(struct pci_dev *dev);
  void (*shutdown)(struct pci_dev *dev);
  int (*sriov_configure)(struct pci_dev *dev, int num_vfs);
  const struct pci_error_handlers *err_handler;
  const struct attribute_group **groups;
  struct device_driver    driver;
  struct pci_dynids       dynids;
};
```

Members

node List of driver structures.

name Driver name.

id_table Pointer to table of device IDs the driver is interested in. Most drivers should export this table using MODULE_DEVICE_TABLE(pci,…).

probe This probing function gets called (during execution of pci_register_driver() for already existing devices or later if a new device gets inserted) for all PCI devices which match the ID table and are not "owned" by the other drivers yet. This function gets passed a "struct pci_dev *" for each device whose entry in the ID table matches the device. The probe function returns zero when the driver chooses to take "ownership" of the device or an error code (negative number) otherwise. The probe function always gets called from process context, so it can sleep.

remove The remove() function gets called whenever a device being handled by this driver is removed (either during deregistration of the driver or when it' s manually pulled out of a hot-pluggable slot). The remove function always gets called from process context, so it can sleep.

suspend Put device into low power state.

resume Wake device from low power state. (Please see Documentation/power/pci.rst for descriptions of PCI Power Management and the related functions.)

shutdown Hook into reboot_notifier_list (kernel/sys.c). Intended to stop any idling DMA operations. Useful for enabling wake-on-lan (NIC) or changing the power state of a device before reboot. e.g. drivers/net/e100.c.

sriov_configure Optional driver callback to allow configuration of number of VFs to enable via sysfs "sriov_numvfs" file.

err_handler See Documentation/PCI/pci-error-recovery.rst

groups Sysfs attribute groups.

driver Driver model structure.

dynids List of dynamically added device IDs.

The ID table is an array of `struct pci_device_id` entries ending with an all-zero entry. Definitions with static const are generally preferred.

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.

Most drivers only need `PCI_DEVICE()` or `PCI_DEVICE_CLASS()` to set up a pci_device_id table.

New PCI IDs may be added to a device driver pci_ids table at runtime as shown below:

```
echo "vendor device subvendor subdevice class class_mask driver_data" > \
/sys/bus/pci/drivers/{driver}/new_id
```

All fields are passed in as hexadecimal values (no leading 0x). The vendor and device fields are mandatory, the others are optional. Users need pass only as many optional fields as necessary:

- subvendor and subdevice fields default to PCI_ANY_ID (FFFFFFFF)
- class and classmask fields default to 0
- driver_data defaults to 0UL.

Note that driver_data must match the value used by any of the pci_device_id entries defined in the driver. This makes the driver_data field mandatory if all the pci_device_id entries have a non-zero driver_data value.

Once added, the driver probe routine will be invoked for any unclaimed PCI devices listed in its (newly updated) pci_ids list.

When the driver exits, it just calls pci_unregister_driver() and the PCI layer automatically calls the remove hook for all devices handled by the driver.

1.2.1 "Attributes" for driver functions/data

Please mark the initialization and cleanup functions where appropriate (the corresponding macros are defined in <linux/init.h>):

__init	Initialization code. Thrown away after the driver initializes.
__exit	Exit code. Ignored for non-modular drivers.

Tips on when/where to use the above attributes:

- The module_init()/module_exit() functions (and all initialization functions called _only_ from these) should be marked __init/__exit.
- Do not mark the struct pci_driver.
- Do NOT mark a function if you are not sure which mark to use. Better to not mark the function than mark the function wrong.

1.3 How to find PCI devices manually

PCI drivers should have a really good reason for not using the pci_register_driver() interface to search for PCI devices. The main reason PCI devices are controlled by multiple drivers is because one PCI device implements several different HW services. E.g. combined serial/parallel port/floppy controller.

A manual search may be performed using the following constructs:

Searching by vendor and device ID:

```
struct pci_dev *dev = NULL;
while (dev = pci_get_device(VENDOR_ID, DEVICE_ID, dev))
        configure_device(dev);
```

Searching by class ID (iterate in a similar way):

```
pci_get_class(CLASS_ID, dev)
```

Searching by both vendor/device and subsystem vendor/device ID:

```
pci_get_subsys(VENDOR_ID,DEVICE_ID, SUBSYS_VENDOR_ID, SUBSYS_DEVICE_ID,␣
↪dev).
```

You can use the constant PCI_ANY_ID as a wildcard replacement for VENDOR_ID or DEVICE_ID. This allows searching for any device from a specific vendor, for example.

These functions are hotplug-safe. They increment the reference count on the pci_dev that they return. You must eventually (possibly at module unload) decrement the reference count on these devices by calling pci_dev_put().

1.4 Device Initialization Steps

As noted in the introduction, most PCI drivers need the following steps for device initialization:

- Enable the device
- Request MMIO/IOP resources
- Set the DMA mask size (for both coherent and streaming DMA)
- Allocate and initialize shared control data (pci_allocate_coherent())
- Access device configuration space (if needed)

- Register IRQ handler (request_irq())
- Initialize non-PCI (i.e. LAN/SCSI/etc parts of the chip)
- Enable DMA/processing engines.

The driver can access PCI config space registers at any time. (Well, almost. When running BIST, config space can go away···but that will just result in a PCI Bus Master Abort and config reads will return garbage).

1.4.1 Enable the PCI device

Before touching any device registers, the driver needs to enable the PCI device by calling pci_enable_device(). This will:

- wake up the device if it was in suspended state,
- allocate I/O and memory regions of the device (if BIOS did not),
- allocate an IRQ (if BIOS did not).

Note: pci_enable_device() can fail! Check the return value.

Warning: OS BUG: we don't check resource allocations before enabling those resources. The sequence would make more sense if we called pci_request_resources() before calling pci_enable_device(). Currently, the device drivers can't detect the bug when when two devices have been allocated the same range. This is not a common problem and unlikely to get fixed soon.

This has been discussed before but not changed as of 2.6.19: http://lkml.org/lkml/2006/3/2/194

pci_set_master() will enable DMA by setting the bus master bit in the PCI_COMMAND register. It also fixes the latency timer value if it's set to something bogus by the BIOS. pci_clear_master() will disable DMA by clearing the bus master bit.

If the PCI device can use the PCI Memory-Write-Invalidate transaction, call pci_set_mwi(). This enables the PCI_COMMAND bit for Mem-Wr-Inval and also ensures that the cache line size register is set correctly. Check the return value of pci_set_mwi() as not all architectures or chip-sets may support Memory-Write-Invalidate. Alternatively, if Mem-Wr-Inval would be nice to have but is not required, call pci_try_set_mwi() to have the system do its best effort at enabling Mem-Wr-Inval.

1.4.2 Request MMIO/IOP resources

Memory (MMIO), and I/O port addresses should NOT be read directly from the PCI device config space. Use the values in the pci_dev structure as the PCI "bus address" might have been remapped to a "host physical" address by the arch/chip-set specific kernel support.

See Documentation/driver-api/io-mapping.rst for how to access device registers or device memory.

The device driver needs to call pci_request_region() to verify no other device is already using the same address resource. Conversely, drivers should call pci_release_region() AFTER calling pci_disable_device(). The idea is to prevent two devices colliding on the same address range.

Tip: See OS BUG comment above. Currently (2.6.19), The driver can only determine MMIO and IO Port resource availability _after_ calling pci_enable_device().

Generic flavors of pci_request_region() are request_mem_region() (for MMIO ranges) and request_region() (for IO Port ranges). Use these for address resources that are not described by "normal" PCI BARs.

Also see pci_request_selected_regions() below.

1.4.3 Set the DMA mask size

Note: If anything below doesn't make sense, please refer to Documentation/DMA-API.txt. This section is just a reminder that drivers need to indicate DMA capabilities of the device and is not an authoritative source for DMA interfaces.

While all drivers should explicitly indicate the DMA capability (e.g. 32 or 64 bit) of the PCI bus master, devices with more than 32-bit bus master capability for streaming data need the driver to "register" this capability by calling pci_set_dma_mask() with appropriate parameters. In general this allows more efficient DMA on systems where System RAM exists above 4G _physical_ address.

Drivers for all PCI-X and PCIe compliant devices must call pci_set_dma_mask() as they are 64-bit DMA devices.

Similarly, drivers must also "register" this capability if the device can directly address "consistent memory" in System RAM above 4G physical address by calling pci_set_consistent_dma_mask(). Again, this includes drivers for all PCI-X and PCIe compliant devices. Many 64-bit "PCI" devices (before PCI-X) and some PCI-X devices are 64-bit DMA capable for payload ("streaming") data but not control ("consistent") data.

1.4.4 Setup shared control data

Once the DMA masks are set, the driver can allocate "consistent" (a.k.a. shared) memory. See Documentation/DMA-API.txt for a full description of the DMA APIs. This section is just a reminder that it needs to be done before enabling DMA on the device.

1.4.5 Initialize device registers

Some drivers will need specific "capability" fields programmed or other "vendor specific" register initialized or reset. E.g. clearing pending interrupts.

1.4.6 Register IRQ handler

While calling request_irq() is the last step described here, this is often just another intermediate step to initialize a device. This step can often be deferred until the device is opened for use.

All interrupt handlers for IRQ lines should be registered with IRQF_SHARED and use the devid to map IRQs to devices (remember that all PCI IRQ lines can be shared).

request_irq() will associate an interrupt handler and device handle with an interrupt number. Historically interrupt numbers represent IRQ lines which run from the PCI device to the Interrupt controller. With MSI and MSI-X (more below) the interrupt number is a CPU "vector".

request_irq() also enables the interrupt. Make sure the device is quiesced and does not have any interrupts pending before registering the interrupt handler.

MSI and MSI-X are PCI capabilities. Both are "Message Signaled Interrupts"which deliver interrupts to the CPU via a DMA write to a Local APIC. The fundamental difference between MSI and MSI-X is how multiple "vectors" get allocated. MSI requires contiguous blocks of vectors while MSI-X can allocate several individual ones.

MSI capability can be enabled by calling pci_alloc_irq_vectors() with the PCI_IRQ_MSI and/or PCI_IRQ_MSIX flags before calling request_irq(). This causes the PCI support to program CPU vector data into the PCI device capability registers. Many architectures, chip-sets, or BIOSes do NOT support MSI or MSI-X and a call to pci_alloc_irq_vectors with just the PCI_IRQ_MSI and PCI_IRQ_MSIX flags will fail, so try to always specify PCI_IRQ_LEGACY as well.

Drivers that have different interrupt handlers for MSI/MSI-X and legacy INTx should chose the right one based on the msi_enabled and msix_enabled flags in the pci_dev structure after calling pci_alloc_irq_vectors.

There are (at least) two really good reasons for using MSI:

1) MSI is an exclusive interrupt vector by definition. This means the interrupt handler doesn't have to verify its device caused the interrupt.

2) MSI avoids DMA/IRQ race conditions. DMA to host memory is guaranteed to be visible to the host CPU(s) when the MSI is delivered. This is important for

both data coherency and avoiding stale control data. This guarantee allows the driver to omit MMIO reads to flush the DMA stream.

See drivers/infiniband/hw/mthca/ or drivers/net/tg3.c for examples of MSI/MSI-X usage.

1.5 PCI device shutdown

When a PCI device driver is being unloaded, most of the following steps need to be performed:

- Disable the device from generating IRQs
- Release the IRQ (free_irq())
- Stop all DMA activity
- Release DMA buffers (both streaming and consistent)
- Unregister from other subsystems (e.g. scsi or netdev)
- Disable device from responding to MMIO/IO Port addresses
- Release MMIO/IO Port resource(s)

1.5.1 Stop IRQs on the device

How to do this is chip/device specific. If it's not done, it opens the possibility of a "screaming interrupt" if (and only if) the IRQ is shared with another device.

When the shared IRQ handler is "unhooked", the remaining devices using the same IRQ line will still need the IRQ enabled. Thus if the "unhooked" device asserts IRQ line, the system will respond assuming it was one of the remaining devices asserted the IRQ line. Since none of the other devices will handle the IRQ, the system will "hang" until it decides the IRQ isn't going to get handled and masks the IRQ (100,000 iterations later). Once the shared IRQ is masked, the remaining devices will stop functioning properly. Not a nice situation.

This is another reason to use MSI or MSI-X if it's available. MSI and MSI-X are defined to be exclusive interrupts and thus are not susceptible to the "screaming interrupt" problem.

1.5.2 Release the IRQ

Once the device is quiesced (no more IRQs), one can call free_irq(). This function will return control once any pending IRQs are handled, "unhook" the drivers IRQ handler from that IRQ, and finally release the IRQ if no one else is using it.

1.5.3 Stop all DMA activity

It's extremely important to stop all DMA operations BEFORE attempting to deallocate DMA control data. Failure to do so can result in memory corruption, hangs, and on some chip-sets a hard crash.

Stopping DMA after stopping the IRQs can avoid races where the IRQ handler might restart DMA engines.

While this step sounds obvious and trivial, several "mature" drivers didn't get this step right in the past.

1.5.4 Release DMA buffers

Once DMA is stopped, clean up streaming DMA first. I.e. unmap data buffers and return buffers to "upstream" owners if there is one.

Then clean up "consistent" buffers which contain the control data.

See Documentation/DMA-API.txt for details on unmapping interfaces.

1.5.5 Unregister from other subsystems

Most low level PCI device drivers support some other subsystem like USB, ALSA, SCSI, NetDev, Infiniband, etc. Make sure your driver isn't losing resources from that other subsystem. If this happens, typically the symptom is an Oops (panic) when the subsystem attempts to call into a driver that has been unloaded.

1.5.6 Disable Device from responding to MMIO/IO Port addresses

io_unmap() MMIO or IO Port resources and then call pci_disable_device(). This is the symmetric opposite of pci_enable_device(). Do not access device registers after calling pci_disable_device().

1.5.7 Release MMIO/IO Port Resource(s)

Call pci_release_region() to mark the MMIO or IO Port range as available. Failure to do so usually results in the inability to reload the driver.

1.6 How to access PCI config space

You can use pci_(read|write)_config_(byte|word|dword) to access the config space of a device represented by struct pci_dev *. All these functions return 0 when successful or an error code (PCIBIOS_…) which can be translated to a text string by pcibios_strerror. Most drivers expect that accesses to valid PCI devices don't fail.

If you don't have a struct pci_dev available, you can call pci_bus_(read|write)_config_(byte|word|dword) to access a given device and function on that bus.

If you access fields in the standard portion of the config header, please use symbolic names of locations and bits declared in <linux/pci.h>.

If you need to access Extended PCI Capability registers, just call pci_find_capability() for the particular capability and it will find the corresponding register block for you.

1.7 Other interesting functions

pci_get_domain_bus_and_slot()	Find pci_dev corresponding to given domain, bus and slot and number. If the device is found, its reference count is increased.
pci_set_power_state()	Set PCI Power Management state (0=D0 ⋯3=D3)
pci_find_capability()	Find specified capability in device’s capability list.
pci_resource_start()	Returns bus start address for a given PCI region
pci_resource_end()	Returns bus end address for a given PCI region
pci_resource_len()	Returns the byte length of a PCI region
pci_set_drvdata()	Set private driver data pointer for a pci_dev
pci_get_drvdata()	Return private driver data pointer for a pci_dev
pci_set_mwi()	Enable Memory-Write-Invalidate transactions.
pci_clear_mwi()	Disable Memory-Write-Invalidate transactions.

1.8 Miscellaneous hints

When displaying PCI device names to the user (for example when a driver wants to tell the user what card has it found), please use pci_name(pci_dev).

Always refer to the PCI devices by a pointer to the pci_dev structure. All PCI layer functions use this identification and it’s the only reasonable one. Don’t use bus/slot/function numbers except for very special purposes – on systems with multiple primary buses their semantics can be pretty complex.

Don’t try to turn on Fast Back to Back writes in your driver. All devices on the bus need to be capable of doing it, so this is something which needs to be handled by platform and generic code, not individual drivers.

1.9 Vendor and device identifications

Do not add new device or vendor IDs to include/linux/pci_ids.h unless they are shared across multiple drivers. You can add private definitions in your driver if they’re helpful, or just use plain hex constants.

The device IDs are arbitrary hex numbers (vendor controlled) and normally used only in a single location, the pci_device_id table.

Please DO submit new vendor/device IDs to http://pci-ids.ucw.cz/. There are mirrors of the pci.ids file at http://pciids.sourceforge.net/ and https://github.com/pciutils/pciids.

1.10 Obsolete functions

There are several functions which you might come across when trying to port an old driver to the new PCI interface. They are no longer present in the kernel as they aren't compatible with hotplug or PCI domains or having sane locking.

pci_find_device()	Superseded by pci_get_device()
pci_find_subsys()	Superseded by pci_get_subsys()
pci_find_slot()	Superseded by pci_get_domain_bus_and_slot()
pci_get_slot()	Superseded by pci_get_domain_bus_and_slot()

The alternative is the traditional PCI device driver that walks PCI device lists. This is still possible but discouraged.

1.11 MMIO Space and "Write Posting"

Converting a driver from using I/O Port space to using MMIO space often requires some additional changes. Specifically, "write posting" needs to be handled. Many drivers (e.g. tg3, acenic, sym53c8xx_2) already do this. I/O Port space guarantees write transactions reach the PCI device before the CPU can continue. Writes to MMIO space allow the CPU to continue before the transaction reaches the PCI device. HW weenies call this "Write Posting" because the write completion is "posted" to the CPU before the transaction has reached its destination.

Thus, timing sensitive code should add readl() where the CPU is expected to wait before doing other work. The classic "bit banging" sequence works fine for I/O Port space:

```
for (i = 8; --i; val >>= 1) {
        outb(val & 1, ioport_reg);      /* write bit */
        udelay(10);
}
```

The same sequence for MMIO space should be:

```
for (i = 8; --i; val >>= 1) {
        writeb(val & 1, mmio_reg);      /* write bit */
        readb(safe_mmio_reg);           /* flush posted write */
        udelay(10);
}
```

It is important that "safe_mmio_reg" not have any side effects that interferes with the correct operation of the device.

Another case to watch out for is when resetting a PCI device. Use PCI Configuration space reads to flush the writel(). This will gracefully handle the PCI master abort on all platforms if the PCI device is expected to not respond to a readl(). Most x86 platforms will allow MMIO reads to master abort (a.k.a. "Soft Fail") and return garbage (e.g. ~0). But many RISC platforms will crash (a.k.a." Hard Fail").

CHAPTER

TWO

THE PCI EXPRESS PORT BUS DRIVER GUIDE HOWTO

Author Tom L Nguyen tom.l.nguyen@intel.com 11/03/2004

2.1 About this guide

This guide describes the basics of the PCI Express Port Bus driver and provides information on how to enable the service drivers to register/unregister with the PCI Express Port Bus Driver.

2.2 What is the PCI Express Port Bus Driver

A PCI Express Port is a logical PCI-PCI Bridge structure. There are two types of PCI Express Port: the Root Port and the Switch Port. The Root Port originates a PCI Express link from a PCI Express Root Complex and the Switch Port connects PCI Express links to internal logical PCI buses. The Switch Port, which has its secondary bus representing the switch's internal routing logic, is called the switch's Upstream Port. The switch's Downstream Port is bridging from switch's internal routing bus to a bus representing the downstream PCI Express link from the PCI Express Switch.

A PCI Express Port can provide up to four distinct functions, referred to in this document as services, depending on its port type. PCI Express Port's services include native hotplug support (HP), power management event support (PME), advanced error reporting support (AER), and virtual channel support (VC). These services may be handled by a single complex driver or be individually distributed and handled by corresponding service drivers.

2.3 Why use the PCI Express Port Bus Driver?

In existing Linux kernels, the Linux Device Driver Model allows a physical device to be handled by only a single driver. The PCI Express Port is a PCI-PCI Bridge device with multiple distinct services. To maintain a clean and simple solution each service may have its own software service driver. In this case several service drivers will compete for a single PCI-PCI Bridge device. For example, if the PCI Express Root Port native hotplug service driver is loaded first, it claims a PCI-PCI Bridge Root Port. The kernel therefore does not load other service drivers for that Root Port. In other words, it is impossible to have multiple service drivers load and run on a PCI-PCI Bridge device simultaneously using the current driver model.

To enable multiple service drivers running simultaneously requires having a PCI Express Port Bus driver, which manages all populated PCI Express Ports and distributes all provided service requests to the corresponding service drivers as required. Some key advantages of using the PCI Express Port Bus driver are listed below:

- Allow multiple service drivers to run simultaneously on a PCI-PCI Bridge Port device.
- Allow service drivers implemented in an independent staged approach.
- Allow one service driver to run on multiple PCI-PCI Bridge Port devices.
- Manage and distribute resources of a PCI-PCI Bridge Port device to requested service drivers.

2.4 Configuring the PCI Express Port Bus Driver vs. Service Drivers

2.4.1 Including the PCI Express Port Bus Driver Support into the Kernel

Including the PCI Express Port Bus driver depends on whether the PCI Express support is included in the kernel config. The kernel will automatically include the PCI Express Port Bus driver as a kernel driver when the PCI Express support is enabled in the kernel.

2.4.2 Enabling Service Driver Support

PCI device drivers are implemented based on Linux Device Driver Model. All service drivers are PCI device drivers. As discussed above, it is impossible to load any service driver once the kernel has loaded the PCI Express Port Bus Driver. To meet the PCI Express Port Bus Driver Model requires some minimal changes on existing service drivers that imposes no impact on the functionality of existing service drivers.

A service driver is required to use the two APIs shown below to register its service with the PCI Express Port Bus driver (see section 5.2.1 & 5.2.2). It is important that a service driver initializes the pcie_port_service_driver data structure, included in

header file /include/linux/pcieport_if.h, before calling these APIs. Failure to do so will result an identity mismatch, which prevents the PCI Express Port Bus driver from loading a service driver.

pcie_port_service_register

```
int pcie_port_service_register(struct pcie_port_service_driver *new)
```

This API replaces the Linux Driver Model's pci_register_driver API. A service driver should always calls pcie_port_service_register at module init. Note that after service driver being loaded, calls such as pci_enable_device(dev) and pci_set_master(dev) are no longer necessary since these calls are executed by the PCI Port Bus driver.

pcie_port_service_unregister

```
void pcie_port_service_unregister(struct pcie_port_service_driver *new)
```

pcie_port_service_unregister replaces the Linux Driver Model's pci_unregister_driver. It's always called by service driver when a module exits.

Sample Code

Below is sample service driver code to initialize the port service driver data structure.

```
static struct pcie_port_service_id service_id[] = { {
  .vendor = PCI_ANY_ID,
  .device = PCI_ANY_ID,
  .port_type = PCIE_RC_PORT,
  .service_type = PCIE_PORT_SERVICE_AER,
  }, { /* end: all zeroes */ }
};

static struct pcie_port_service_driver root_aerdrv = {
  .name               = (char *)device_name,
  .id_table   = &service_id[0],

  .probe              = aerdrv_load,
  .remove             = aerdrv_unload,

  .suspend    = aerdrv_suspend,
  .resume             = aerdrv_resume,
};
```

Below is a sample code for registering/unregistering a service driver.

```
static int __init aerdrv_service_init(void)
{
  int retval = 0;
```

(continues on next page)

(continued from previous page)

```
  retval = pcie_port_service_register(&root_aerdrv);
  if (!retval) {
    /*
    * FIX ME
    */
  }
  return retval;
}

static void __exit aerdrv_service_exit(void)
{
  pcie_port_service_unregister(&root_aerdrv);
}

module_init(aerdrv_service_init);
module_exit(aerdrv_service_exit);
```

2.5 Possible Resource Conflicts

Since all service drivers of a PCI-PCI Bridge Port device are allowed to run simultaneously, below lists a few of possible resource conflicts with proposed solutions.

2.5.1 MSI and MSI-X Vector Resource

Once MSI or MSI-X interrupts are enabled on a device, it stays in this mode until they are disabled again. Since service drivers of the same PCI-PCI Bridge port share the same physical device, if an individual service driver enables or disables MSI/MSI-X mode it may result unpredictable behavior.

To avoid this situation all service drivers are not permitted to switch interrupt mode on its device. The PCI Express Port Bus driver is responsible for determining the interrupt mode and this should be transparent to service drivers. Service drivers need to know only the vector IRQ assigned to the field irq of struct pcie_device, which is passed in when the PCI Express Port Bus driver probes each service driver. Service drivers should use (struct pcie_device*)dev->irq to call request_irq/free_irq. In addition, the interrupt mode is stored in the field interrupt_mode of struct pcie_device.

2.5.2 PCI Memory/IO Mapped Regions

Service drivers for PCI Express Power Management (PME), Advanced Error Reporting (AER), Hot-Plug (HP) and Virtual Channel (VC) access PCI configuration space on the PCI Express port. In all cases the registers accessed are independent of each other. This patch assumes that all service drivers will be well behaved and not overwrite other service driver' s configuration settings.

2.5.3 PCI Config Registers

Each service driver runs its PCI config operations on its own capability structure except the PCI Express capability structure, in which Root Control register and Device Control register are shared between PME and AER. This patch assumes that all service drivers will be well behaved and not overwrite other service driver' s configuration settings.

CHAPTER

THREE

PCI EXPRESS I/O VIRTUALIZATION HOWTO

Authors

- Yu Zhao <yu.zhao@intel.com>
- Donald Dutile <ddutile@redhat.com>

3.1 Overview

3.1.1 What is SR-IOV

Single Root I/O Virtualization (SR-IOV) is a PCI Express Extended capability which makes one physical device appear as multiple virtual devices. The physical device is referred to as Physical Function (PF) while the virtual devices are referred to as Virtual Functions (VF). Allocation of the VF can be dynamically controlled by the PF via registers encapsulated in the capability. By default, this feature is not enabled and the PF behaves as traditional PCIe device. Once it' s turned on, each VF' s PCI configuration space can be accessed by its own Bus, Device and Function Number (Routing ID). And each VF also has PCI Memory Space, which is used to map its register set. VF device driver operates on the register set so it can be functional and appear as a real existing PCI device.

3.2 User Guide

3.2.1 How can I enable SR-IOV capability

Multiple methods are available for SR-IOV enablement. In the first method, the device driver (PF driver) will control the enabling and disabling of the capability via API provided by SR-IOV core. If the hardware has SR-IOV capability, loading its PF driver would enable it and all VFs associated with the PF. Some PF drivers require a module parameter to be set to determine the number of VFs to enable. In the second method, a write to the sysfs file sriov_numvfs will enable and disable the VFs associated with a PCIe PF. This method enables per-PF, VF enable/disable values versus the first method, which applies to all PFs of the same device. Additionally, the PCI SRIOV core support ensures that enable/disable operations are valid to reduce duplication in multiple drivers for the same checks, e.g., check

numvfs == 0 if enabling VFs, ensure numvfs <= totalvfs. The second method is the recommended method for new/future VF devices.

3.2.2 How can I use the Virtual Functions

The VF is treated as hot-plugged PCI devices in the kernel, so they should be able to work in the same way as real PCI devices. The VF requires device driver that is same as a normal PCI device’s.

3.3 Developer Guide

3.3.1 SR-IOV API

To enable SR-IOV capability:

(a) For the first method, in the driver:

```
int pci_enable_sriov(struct pci_dev *dev, int nr_virtfn);
```

‘nr_virtfn’ is number of VFs to be enabled.

(b) For the second method, from sysfs:

```
echo 'nr_virtfn' > \
/sys/bus/pci/devices/<DOMAIN:BUS:DEVICE.FUNCTION>/sriov_numvfs
```

To disable SR-IOV capability:

(a) For the first method, in the driver:

```
void pci_disable_sriov(struct pci_dev *dev);
```

(b) For the second method, from sysfs:

```
echo  0 > \
/sys/bus/pci/devices/<DOMAIN:BUS:DEVICE.FUNCTION>/sriov_numvfs
```

To enable auto probing VFs by a compatible driver on the host, run command below before enabling SR-IOV capabilities. This is the default behavior.

```
echo 1 > \
/sys/bus/pci/devices/<DOMAIN:BUS:DEVICE.FUNCTION>/sriov_drivers_autoprobe
```

To disable auto probing VFs by a compatible driver on the host, run command below before enabling SR-IOV capabilities. Updating this entry will not affect VFs which are already probed.

```
echo  0 > \
/sys/bus/pci/devices/<DOMAIN:BUS:DEVICE.FUNCTION>/sriov_drivers_autoprobe
```

3.3.2 Usage example

Following piece of code illustrates the usage of the SR-IOV API.

```
static int dev_probe(struct pci_dev *dev, const struct pci_device_id *id)
{
        pci_enable_sriov(dev, NR_VIRTFN);

        ...

        return 0;
}

static void dev_remove(struct pci_dev *dev)
{
        pci_disable_sriov(dev);

        ...
}

static int dev_suspend(struct pci_dev *dev, pm_message_t state)
{
        ...

        return 0;
}

static int dev_resume(struct pci_dev *dev)
{
        ...

        return 0;
}

static void dev_shutdown(struct pci_dev *dev)
{
        ...
}

static int dev_sriov_configure(struct pci_dev *dev, int numvfs)
{
        if (numvfs > 0) {
                ...
                pci_enable_sriov(dev, numvfs);
                ...
                return numvfs;
        }
        if (numvfs == 0) {
                ....
                pci_disable_sriov(dev);
                ...
                return 0;
        }
}

static struct pci_driver dev_driver = {
        .name =         "SR-IOV Physical Function driver",
```

(continues on next page)

(continued from previous page)

```
        .id_table =     dev_id_table,
        .probe =        dev_probe,
        .remove =       dev_remove,
        .suspend =      dev_suspend,
        .resume =       dev_resume,
        .shutdown =     dev_shutdown,
        .sriov_configure = dev_sriov_configure,
};
```

CHAPTER

FOUR

THE MSI DRIVER GUIDE HOWTO

Authors Tom L Nguyen; Martine Silbermann; Matthew Wilcox

4.1 About this guide

This guide describes the basics of Message Signaled Interrupts (MSIs), the advantages of using MSI over traditional interrupt mechanisms, how to change your driver to use MSI or MSI-X and some basic diagnostics to try if a device doesn't support MSIs.

4.2 What are MSIs?

A Message Signaled Interrupt is a write from the device to a special address which causes an interrupt to be received by the CPU.

The MSI capability was first specified in PCI 2.2 and was later enhanced in PCI 3.0 to allow each interrupt to be masked individually. The MSI-X capability was also introduced with PCI 3.0. It supports more interrupts per device than MSI and allows interrupts to be independently configured.

Devices may support both MSI and MSI-X, but only one can be enabled at a time.

4.3 Why use MSIs?

There are three reasons why using MSIs can give an advantage over traditional pin-based interrupts.

Pin-based PCI interrupts are often shared amongst several devices. To support this, the kernel must call each interrupt handler associated with an interrupt, which leads to reduced performance for the system as a whole. MSIs are never shared, so this problem cannot arise.

When a device writes data to memory, then raises a pin-based interrupt, it is possible that the interrupt may arrive before all the data has arrived in memory (this becomes more likely with devices behind PCI-PCI bridges). In order to ensure that all the data has arrived in memory, the interrupt handler must read a register on the device which raised the interrupt. PCI transaction ordering rules require that

all the data arrive in memory before the value may be returned from the register. Using MSIs avoids this problem as the interrupt-generating write cannot pass the data writes, so by the time the interrupt is raised, the driver knows that all the data has arrived in memory.

PCI devices can only support a single pin-based interrupt per function. Often drivers have to query the device to find out what event has occurred, slowing down interrupt handling for the common case. With MSIs, a device can support more interrupts, allowing each interrupt to be specialised to a different purpose. One possible design gives infrequent conditions (such as errors) their own interrupt which allows the driver to handle the normal interrupt handling path more efficiently. Other possible designs include giving one interrupt to each packet queue in a network card or each port in a storage controller.

4.4 How to use MSIs

PCI devices are initialised to use pin-based interrupts. The device driver has to set up the device to use MSI or MSI-X. Not all machines support MSIs correctly, and for those machines, the APIs described below will simply fail and the device will continue to use pin-based interrupts.

4.4.1 Include kernel support for MSIs

To support MSI or MSI-X, the kernel must be built with the CONFIG_PCI_MSI option enabled. This option is only available on some architectures, and it may depend on some other options also being set. For example, on x86, you must also enable X86_UP_APIC or SMP in order to see the CONFIG_PCI_MSI option.

4.4.2 Using MSI

Most of the hard work is done for the driver in the PCI layer. The driver simply has to request that the PCI layer set up the MSI capability for this device.

To automatically use MSI or MSI-X interrupt vectors, use the following function:

```
int pci_alloc_irq_vectors(struct pci_dev *dev, unsigned int min_vecs,
              unsigned int max_vecs, unsigned int flags);
```

which allocates up to max_vecs interrupt vectors for a PCI device. It returns the number of vectors allocated or a negative error. If the device has a requirements for a minimum number of vectors the driver can pass a min_vecs argument set to this limit, and the PCI core will return -ENOSPC if it can't meet the minimum number of vectors.

The flags argument is used to specify which type of interrupt can be used by the device and the driver (PCI_IRQ_LEGACY, PCI_IRQ_MSI, PCI_IRQ_MSIX). A convenient short-hand (PCI_IRQ_ALL_TYPES) is also available to ask for any possible kind of interrupt. If the PCI_IRQ_AFFINITY flag is set, pci_alloc_irq_vectors() will spread the interrupts around the available CPUs.

To get the Linux IRQ numbers passed to request_irq() and free_irq() and the vectors, use the following function:

```
int pci_irq_vector(struct pci_dev *dev, unsigned int nr);
```

Any allocated resources should be freed before removing the device using the following function:

```
void pci_free_irq_vectors(struct pci_dev *dev);
```

If a device supports both MSI-X and MSI capabilities, this API will use the MSI-X facilities in preference to the MSI facilities. MSI-X supports any number of interrupts between 1 and 2048. In contrast, MSI is restricted to a maximum of 32 interrupts (and must be a power of two). In addition, the MSI interrupt vectors must be allocated consecutively, so the system might not be able to allocate as many vectors for MSI as it could for MSI-X. On some platforms, MSI interrupts must all be targeted at the same set of CPUs whereas MSI-X interrupts can all be targeted at different CPUs.

If a device supports neither MSI-X or MSI it will fall back to a single legacy IRQ vector.

The typical usage of MSI or MSI-X interrupts is to allocate as many vectors as possible, likely up to the limit supported by the device. If nvec is larger than the number supported by the device it will automatically be capped to the supported limit, so there is no need to query the number of vectors supported beforehand:

```
nvec = pci_alloc_irq_vectors(pdev, 1, nvec, PCI_IRQ_ALL_TYPES)
if (nvec < 0)
        goto out_err;
```

If a driver is unable or unwilling to deal with a variable number of MSI interrupts it can request a particular number of interrupts by passing that number to pci_alloc_irq_vectors() function as both 'min_vecs' and 'max_vecs' parameters:

```
ret = pci_alloc_irq_vectors(pdev, nvec, nvec, PCI_IRQ_ALL_TYPES);
if (ret < 0)
        goto out_err;
```

The most notorious example of the request type described above is enabling the single MSI mode for a device. It could be done by passing two 1s as 'min_vecs' and 'max_vecs':

```
ret = pci_alloc_irq_vectors(pdev, 1, 1, PCI_IRQ_ALL_TYPES);
if (ret < 0)
        goto out_err;
```

Some devices might not support using legacy line interrupts, in which case the driver can specify that only MSI or MSI-X is acceptable:

```
nvec = pci_alloc_irq_vectors(pdev, 1, nvec, PCI_IRQ_MSI | PCI_IRQ_MSIX);
if (nvec < 0)
        goto out_err;
```

4.4.3 Legacy APIs

The following old APIs to enable and disable MSI or MSI-X interrupts should not be used in new code:

```
pci_enable_msi()              /* deprecated */
pci_disable_msi()             /* deprecated */
pci_enable_msix_range()       /* deprecated */
pci_enable_msix_exact()       /* deprecated */
pci_disable_msix()            /* deprecated */
```

Additionally there are APIs to provide the number of supported MSI or MSI-X vectors: pci_msi_vec_count() and pci_msix_vec_count(). In general these should be avoided in favor of letting pci_alloc_irq_vectors() cap the number of vectors. If you have a legitimate special use case for the count of vectors we might have to revisit that decision and add a pci_nr_irq_vectors() helper that handles MSI and MSI-X transparently.

4.4.4 Considerations when using MSIs

Spinlocks

Most device drivers have a per-device spinlock which is taken in the interrupt handler. With pin-based interrupts or a single MSI, it is not necessary to disable interrupts (Linux guarantees the same interrupt will not be re-entered). If a device uses multiple interrupts, the driver must disable interrupts while the lock is held. If the device sends a different interrupt, the driver will deadlock trying to recursively acquire the spinlock. Such deadlocks can be avoided by using spin_lock_irqsave() or spin_lock_irq() which disable local interrupts and acquire the lock (see Documentation/kernel-hacking/locking.rst).

4.4.5 How to tell whether MSI/MSI-X is enabled on a device

Using ‘lspci -v’ (as root) may show some devices with “MSI”, “Message Signalled Interrupts” or “MSI-X” capabilities. Each of these capabilities has an ‘Enable’ flag which is followed with either “+” (enabled) or “-” (disabled).

4.5 MSI quirks

Several PCI chipsets or devices are known not to support MSIs. The PCI stack provides three ways to disable MSIs:

1. globally
2. on all devices behind a specific bridge
3. on a single device

4.5.1 Disabling MSIs globally

Some host chipsets simply don't support MSIs properly. If we're lucky, the manufacturer knows this and has indicated it in the ACPI FADT table. In this case, Linux automatically disables MSIs. Some boards don't include this information in the table and so we have to detect them ourselves. The complete list of these is found near the quirk_disable_all_msi() function in drivers/pci/quirks.c.

If you have a board which has problems with MSIs, you can pass pci=nomsi on the kernel command line to disable MSIs on all devices. It would be in your best interests to report the problem to linux-pci@vger.kernel.org including a full 'lspci -v' so we can add the quirks to the kernel.

4.5.2 Disabling MSIs below a bridge

Some PCI bridges are not able to route MSIs between busses properly. In this case, MSIs must be disabled on all devices behind the bridge.

Some bridges allow you to enable MSIs by changing some bits in their PCI configuration space (especially the Hypertransport chipsets such as the nVidia nForce and Serverworks HT2000). As with host chipsets, Linux mostly knows about them and automatically enables MSIs if it can. If you have a bridge unknown to Linux, you can enable MSIs in configuration space using whatever method you know works, then enable MSIs on that bridge by doing:

```
echo 1 > /sys/bus/pci/devices/$bridge/msi_bus
```

where $bridge is the PCI address of the bridge you've enabled (eg 0000:00:0e.0).

To disable MSIs, echo 0 instead of 1. Changing this value should be done with caution as it could break interrupt handling for all devices below this bridge.

Again, please notify linux-pci@vger.kernel.org of any bridges that need special handling.

4.5.3 Disabling MSIs on a single device

Some devices are known to have faulty MSI implementations. Usually this is handled in the individual device driver, but occasionally it's necessary to handle this with a quirk. Some drivers have an option to disable use of MSI. While this is a convenient workaround for the driver author, it is not good practice, and should not be emulated.

4.5.4 Finding why MSIs are disabled on a device

From the above three sections, you can see that there are many reasons why MSIs may not be enabled for a given device. Your first step should be to examine your dmesg carefully to determine whether MSIs are enabled for your machine. You should also check your .config to be sure you have enabled CONFIG_PCI_MSI.

Then, 'lspci -t' gives the list of bridges above a device. Reading /sys/bus/pci/devices/*/msi_bus will tell you whether MSIs are enabled (1) or disabled (0). If 0 is found in any of the msi_bus files belonging to bridges between the PCI root and the device, MSIs are disabled.

It is also worth checking the device driver to see whether it supports MSIs. For example, it may contain calls to pci_alloc_irq_vectors() with the PCI_IRQ_MSI or PCI_IRQ_MSIX flags.

CHAPTER

FIVE

ACPI CONSIDERATIONS FOR PCI HOST BRIDGES

The general rule is that the ACPI namespace should describe everything the OS might use unless there's another way for the OS to find it [1, 2].

For example, there's no standard hardware mechanism for enumerating PCI host bridges, so the ACPI namespace must describe each host bridge, the method for accessing PCI config space below it, the address space windows the host bridge forwards to PCI (using _CRS), and the routing of legacy INTx interrupts (using _PRT).

PCI devices, which are below the host bridge, generally do not need to be described via ACPI. The OS can discover them via the standard PCI enumeration mechanism, using config accesses to discover and identify devices and read and size their BARs. However, ACPI may describe PCI devices if it provides power management or hotplug functionality for them or if the device has INTx interrupts connected by platform interrupt controllers and a _PRT is needed to describe those connections.

ACPI resource description is done via _CRS objects of devices in the ACPI namespace [2]. The _CRS is like a generalized PCI BAR: the OS can read _CRS and figure out what resource is being consumed even if it doesn't have a driver for the device [3]. That's important because it means an old OS can work correctly even on a system with new devices unknown to the OS. The new devices might not do anything, but the OS can at least make sure no resources conflict with them.

Static tables like MCFG, HPET, ECDT, etc., are not mechanisms for reserving address space. The static tables are for things the OS needs to know early in boot, before it can parse the ACPI namespace. If a new table is defined, an old OS needs to operate correctly even though it ignores the table. _CRS allows that because it is generic and understood by the old OS; a static table does not.

If the OS is expected to manage a non-discoverable device described via ACPI, that device will have a specific _HID/_CID that tells the OS what driver to bind to it, and the _CRS tells the OS and the driver where the device's registers are.

PCI host bridges are PNP0A03 or PNP0A08 devices. Their _CRS should describe all the address space they consume. This includes all the windows they forward down to the PCI bus, as well as registers of the host bridge itself that are not forwarded to PCI. The host bridge registers include things like secondary/subordinate bus registers that determine the bus range below the bridge, window registers that describe the apertures, etc. These are all device-specific, non-architected things, so the only way a PNP0A03/PNP0A08 driver can manage

them is via _PRS/_CRS/_SRS, which contain the device-specific details. The host bridge registers also include ECAM space, since it is consumed by the host bridge.

ACPI defines a Consumer/Producer bit to distinguish the bridge registers ("Consumer") from the bridge apertures ("Producer") [4, 5], but early BIOSes didn't use that bit correctly. The result is that the current ACPI spec defines Consumer/Producer only for the Extended Address Space descriptors; the bit should be ignored in the older QWord/DWord/Word Address Space descriptors. Consequently, OSes have to assume all QWord/DWord/Word descriptors are windows.

Prior to the addition of Extended Address Space descriptors, the failure of Consumer/Producer meant there was no way to describe bridge registers in the PNP0A03/PNP0A08 device itself. The workaround was to describe the bridge registers (including ECAM space) in PNP0C02 catch-all devices [6]. With the exception of ECAM, the bridge register space is device-specific anyway, so the generic PNP0A03/PNP0A08 driver (pci_root.c) has no need to know about it.

New architectures should be able to use "Consumer" Extended Address Space descriptors in the PNP0A03 device for bridge registers, including ECAM, although a strict interpretation of [6] might prohibit this. Old x86 and ia64 kernels assume all address space descriptors, including "Consumer" Extended Address Space ones, are windows, so it would not be safe to describe bridge registers this way on those architectures.

PNP0C02 "motherboard" devices are basically a catch-all. There's no programming model for them other than "don't use these resources for anything else." So a PNP0C02 _CRS should claim any address space that is (1) not claimed by _CRS under any other device object in the ACPI namespace and (2) should not be assigned by the OS to something else.

The PCIe spec requires the Enhanced Configuration Access Method (ECAM) unless there's a standard firmware interface for config access, e.g., the ia64 SAL interface [7]. A host bridge consumes ECAM memory address space and converts memory accesses into PCI configuration accesses. The spec defines the ECAM address space layout and functionality; only the base of the address space is device-specific. An ACPI OS learns the base address from either the static MCFG table or a _CBA method in the PNP0A03 device.

The MCFG table must describe the ECAM space of non-hot pluggable host bridges [8]. Since MCFG is a static table and can't be updated by hotplug, a _CBA method in the PNP0A03 device describes the ECAM space of a hot-pluggable host bridge [9]. Note that for both MCFG and _CBA, the base address always corresponds to bus 0, even if the bus range below the bridge (which is reported via _CRS) doesn't start at 0.

[1] ACPI 6.2, sec 6.1: For any device that is on a non-enumerable type of bus (for example, an ISA bus), OSPM enumerates the devices' identifier(s) and the ACPI system firmware must supply an _HID object ⋯for each device to enable OSPM to do that.

[2] ACPI 6.2, sec 3.7: The OS enumerates motherboard devices simply by reading through the ACPI Namespace looking for devices with hardware IDs.

Each device enumerated by ACPI includes ACPI-defined objects in the ACPI Namespace that report the hardware resources the device could occupy [_PRS], an object that reports the resources that are currently used by the

device [_CRS], and objects for configuring those resources [_SRS]. The information is used by the Plug and Play OS (OSPM) to configure the devices.

[3] ACPI 6.2, sec 6.2: OSPM uses device configuration objects to configure hardware resources for devices enumerated via ACPI. Device configuration objects provide information about current and possible resource requirements, the relationship between shared resources, and methods for configuring hardware resources.

When OSPM enumerates a device, it calls _PRS to determine the resource requirements of the device. It may also call _CRS to find the current resource settings for the device. Using this information, the Plug and Play system determines what resources the device should consume and sets those resources by calling the device's _SRS control method.

In ACPI, devices can consume resources (for example, legacy keyboards), provide resources (for example, a proprietary PCI bridge), or do both. Unless otherwise specified, resources for a device are assumed to be taken from the nearest matching resource above the device in the device hierarchy.

[4] ACPI 6.2, sec 6.4.3.5.1, 2, 3, 4:

QWord/DWord/Word Address Space Descriptor (.1, .2, .3) General Flags: Bit [0] Ignored

Extended Address Space Descriptor (.4) General Flags: Bit [0] Consumer/Producer:

- 1 – This device consumes this resource
- 0 – This device produces and consumes this resource

[5] ACPI 6.2, sec 19.6.43: ResourceUsage specifies whether the Memory range is consumed by this device (ResourceConsumer) or passed on to child devices (ResourceProducer). If nothing is specified, then ResourceConsumer is assumed.

[6] PCI Firmware 3.2, sec 4.1.2: If the operating system does not natively comprehend reserving the MMCFG region, the MMCFG region must be reserved by firmware. The address range reported in the MCFG table or by _CBA method (see Section 4.1.3) must be reserved by declaring a motherboard resource. For most systems, the motherboard resource would appear at the root of the ACPI namespace (under _SB) in a node with a _HID of EISAID (PNP0C02), and the resources in this case should not be claimed in the root PCI bus's _CRS. The resources can optionally be returned in Int15 E820 or EFIGetMemoryMap as reserved memory but must always be reported through ACPI as a motherboard resource.

[7] PCI Express 4.0, sec 7.2.2: For systems that are PC-compatible, or that do not implement a processor-architecture-specific firmware interface standard that allows access to the Configuration Space, the ECAM is required as defined in this section.

[8] PCI Firmware 3.2, sec 4.1.2: The MCFG table is an ACPI table that is used to communicate the base addresses corresponding to the non-hot removable PCI Segment Groups range within a PCI Segment Group available to the operating system at boot. This is required for the PC-compatible systems.

The MCFG table is only used to communicate the base addresses corresponding to the PCI Segment Groups available to the system at boot.

[9] PCI Firmware 3.2, sec 4.1.3: The _CBA (Memory mapped Configuration Base Address) control method is an optional ACPI object that returns the 64-bit memory mapped configuration base address for the hot plug capable host bridge. The base address returned by _CBA is processor-relative address. The _CBA control method evaluates to an Integer.

This control method appears under a host bridge object. When the _CBA method appears under an active host bridge object, the operating system evaluates this structure to identify the memory mapped configuration base address corresponding to the PCI Segment Group for the bus number range specified in _CRS method. An ACPI name space object that contains the _CBA method must also contain a corresponding _SEG method.

CHAPTER

SIX

PCI ERROR RECOVERY

Authors

- Linas Vepstas <linasvepstas@gmail.com>
- Richard Lary <rlary@us.ibm.com>
- Mike Mason <mmlnx@us.ibm.com>

Many PCI bus controllers are able to detect a variety of hardware PCI errors on the bus, such as parity errors on the data and address buses, as well as SERR and PERR errors. Some of the more advanced chipsets are able to deal with these errors; these include PCI-E chipsets, and the PCI-host bridges found on IBM Power4, Power5 and Power6-based pSeries boxes. A typical action taken is to disconnect the affected device, halting all I/O to it. The goal of a disconnection is to avoid system corruption; for example, to halt system memory corruption due to DMA's to "wild" addresses. Typically, a reconnection mechanism is also offered, so that the affected PCI device(s) are reset and put back into working condition. The reset phase requires coordination between the affected device drivers and the PCI controller chip. This document describes a generic API for notifying device drivers of a bus disconnection, and then performing error recovery. This API is currently implemented in the 2.6.16 and later kernels.

Reporting and recovery is performed in several steps. First, when a PCI hardware error has resulted in a bus disconnect, that event is reported as soon as possible to all affected device drivers, including multiple instances of a device driver on multi-function cards. This allows device drivers to avoid deadlocking in spinloops, waiting for some i/o-space register to change, when it never will. It also gives the drivers a chance to defer incoming I/O as needed.

Next, recovery is performed in several stages. Most of the complexity is forced by the need to handle multi-function devices, that is, devices that have multiple device drivers associated with them. In the first stage, each driver is allowed to indicate what type of reset it desires, the choices being a simple re-enabling of I/O or requesting a slot reset.

If any driver requests a slot reset, that is what will be done.

After a reset and/or a re-enabling of I/O, all drivers are again notified, so that they may then perform any device setup/config that may be required. After these have all completed, a final "resume normal operations" event is sent out.

The biggest reason for choosing a kernel-based implementation rather than a user-space implementation was the need to deal with bus disconnects of PCI devices attached to storage media, and, in particular, disconnects from devices holding

the root file system. If the root file system is disconnected, a user-space mechanism would have to go through a large number of contortions to complete recovery. Almost all of the current Linux file systems are not tolerant of disconnection from/reconnection to their underlying block device. By contrast, bus errors are easy to manage in the device driver. Indeed, most device drivers already handle very similar recovery procedures; for example, the SCSI-generic layer already provides significant mechanisms for dealing with SCSI bus errors and SCSI bus resets.

6.1 Detailed Design

Design and implementation details below, based on a chain of public email discussions with Ben Herrenschmidt, circa 5 April 2005.

The error recovery API support is exposed to the driver in the form of a structure of function pointers pointed to by a new field in struct pci_driver. A driver that fails to provide the structure is "non-aware", and the actual recovery steps taken are platform dependent. The arch/powerpc implementation will simulate a PCI hotplug remove/add.

This structure has the form:

```
struct pci_error_handlers
{
        int (*error_detected)(struct pci_dev *dev, enum pci_channel_state);
        int (*mmio_enabled)(struct pci_dev *dev);
        int (*slot_reset)(struct pci_dev *dev);
        void (*resume)(struct pci_dev *dev);
};
```

The possible channel states are:

```
enum pci_channel_state {
        pci_channel_io_normal,  /* I/O channel is in normal state */
        pci_channel_io_frozen,  /* I/O to channel is blocked */
        pci_channel_io_perm_failure, /* PCI card is dead */
};
```

Possible return values are:

```
enum pci_ers_result {
        PCI_ERS_RESULT_NONE,        /* no result/none/not supported in␣
↪device driver */
        PCI_ERS_RESULT_CAN_RECOVER, /* Device driver can recover without␣
↪slot reset */
        PCI_ERS_RESULT_NEED_RESET,  /* Device driver wants slot to be␣
↪reset. */
        PCI_ERS_RESULT_DISCONNECT,  /* Device has completely failed, is␣
↪unrecoverable */
        PCI_ERS_RESULT_RECOVERED,   /* Device driver is fully recovered␣
↪and operational */
};
```

A driver does not have to implement all of these callbacks; however, if it implements any, it must implement error_detected(). If a callback is not imple-

mented, the corresponding feature is considered unsupported. For example, if mmio_enabled() and resume() aren't there, then it is assumed that the driver is not doing any direct recovery and requires a slot reset. Typically a driver will want to know about a slot_reset().

The actual steps taken by a platform to recover from a PCI error event will be platform-dependent, but will follow the general sequence described below.

6.1.1 STEP 0: Error Event

A PCI bus error is detected by the PCI hardware. On powerpc, the slot is isolated, in that all I/O is blocked: all reads return 0xffffffff, all writes are ignored.

6.1.2 STEP 1: Notification

Platform calls the error_detected() callback on every instance of every driver affected by the error.

At this point, the device might not be accessible anymore, depending on the platform (the slot will be isolated on powerpc). The driver may already have "noticed" the error because of a failing I/O, but this is the proper "synchronization point", that is, it gives the driver a chance to cleanup, waiting for pending stuff (timers, whatever, etc…) to complete; it can take semaphores, schedule, etc…everything but touch the device. Within this function and after it returns, the driver shouldn't do any new IOs. Called in task context. This is sort of a "quiesce" point. See note about interrupts at the end of this doc.

All drivers participating in this system must implement this call. The driver must return one of the following result codes:

- **PCI_ERS_RESULT_CAN_RECOVER** Driver returns this if it thinks it might be able to recover the HW by just banging IOs or if it wants to be given a chance to extract some diagnostic information (see mmio_enable, below).
- **PCI_ERS_RESULT_NEED_RESET** Driver returns this if it can't recover without a slot reset.
- **PCI_ERS_RESULT_DISCONNECT** Driver returns this if it doesn't want to recover at all.

The next step taken will depend on the result codes returned by the drivers.

If all drivers on the segment/slot return PCI_ERS_RESULT_CAN_RECOVER, then the platform should re-enable IOs on the slot (or do nothing in particular, if the platform doesn't isolate slots), and recovery proceeds to STEP 2 (MMIO Enable).

If any driver requested a slot reset (by returning PCI_ERS_RESULT_NEED_RESET), then recovery proceeds to STEP 4 (Slot Reset).

If the platform is unable to recover the slot, the next step is STEP 6 (Permanent Failure).

Note: The current powerpc implementation assumes that a device driver will

not schedule or semaphore in this routine; the current powerpc implementation uses one kernel thread to notify all devices; thus, if one device sleeps/schedules, all devices are affected. Doing better requires complex multi-threaded logic in the error recovery implementation (e.g. waiting for all notification threads to "join" before proceeding with recovery.) This seems excessively complex and not worth implementing.

The current powerpc implementation doesn't much care if the device attempts I/O at this point, or not. I/O's will fail, returning a value of 0xff on read, and writes will be dropped. If more than EEH_MAX_FAILS I/O's are attempted to a frozen adapter, EEH assumes that the device driver has gone into an infinite loop and prints an error to syslog. A reboot is then required to get the device working again.

6.1.3 STEP 2: MMIO Enabled

The platform re-enables MMIO to the device (but typically not the DMA), and then calls the mmio_enabled() callback on all affected device drivers.

This is the "early recovery" call. IOs are allowed again, but DMA is not, with some restrictions. This is NOT a callback for the driver to start operations again, only to peek/poke at the device, extract diagnostic information, if any, and eventually do things like trigger a device local reset or some such, but not restart operations. This callback is made if all drivers on a segment agree that they can try to recover and if no automatic link reset was performed by the HW. If the platform can't just re-enable IOs without a slot reset or a link reset, it will not call this callback, and instead will have gone directly to STEP 3 (Link Reset) or STEP 4 (Slot Reset)

Note: The following is proposed; no platform implements this yet: Proposal: All I/O's should be done _synchronously_ from within this callback, errors triggered by them will be returned via the normal pci_check_whatever() API, no new error_detected() callback will be issued due to an error happening here. However, such an error might cause IOs to be re-blocked for the whole segment, and thus invalidate the recovery that other devices on the same segment might have done, forcing the whole segment into one of the next states, that is, link reset or slot reset.

The driver should return one of the following result codes:

- **PCI_ERS_RESULT_RECOVERED** Driver returns this if it thinks the device is fully functional and thinks it is ready to start normal driver operations again. There is no guarantee that the driver will actually be allowed to proceed, as another driver on the same segment might have failed and thus triggered a slot reset on platforms that support it.
- **PCI_ERS_RESULT_NEED_RESET** Driver returns this if it thinks the device is not recoverable in its current state and it needs a slot reset to proceed.

- **PCI_ERS_RESULT_DISCONNECT** Same as above. Total failure, no recovery even after reset driver dead. (To be defined more precisely)

The next step taken depends on the results returned by the drivers. If all drivers returned PCI_ERS_RESULT_RECOVERED, then the platform proceeds to either STEP3 (Link Reset) or to STEP 5 (Resume Operations).

If any driver returned PCI_ERS_RESULT_NEED_RESET, then the platform proceeds to STEP 4 (Slot Reset)

6.1.4 STEP 3: Link Reset

The platform resets the link. This is a PCI-Express specific step and is done whenever a fatal error has been detected that can be "solved" by resetting the link.

6.1.5 STEP 4: Slot Reset

In response to a return value of PCI_ERS_RESULT_NEED_RESET, the the platform will perform a slot reset on the requesting PCI device(s). The actual steps taken by a platform to perform a slot reset will be platform-dependent. Upon completion of slot reset, the platform will call the device slot_reset() callback.

Powerpc platforms implement two levels of slot reset: soft reset(default) and fundamental(optional) reset.

Powerpc soft reset consists of asserting the adapter #RST line and then restoring the PCI BAR's and PCI configuration header to a state that is equivalent to what it would be after a fresh system power-on followed by power-on BIOS/system firmware initialization. Soft reset is also known as hot-reset.

Powerpc fundamental reset is supported by PCI Express cards only and results in device's state machines, hardware logic, port states and configuration registers to initialize to their default conditions.

For most PCI devices, a soft reset will be sufficient for recovery. Optional fundamental reset is provided to support a limited number of PCI Express devices for which a soft reset is not sufficient for recovery.

If the platform supports PCI hotplug, then the reset might be performed by toggling the slot electrical power off/on.

It is important for the platform to restore the PCI config space to the "fresh poweron"state, rather than the "last state". After a slot reset, the device driver will almost always use its standard device initialization routines, and an unusual config space setup may result in hung devices, kernel panics, or silent data corruption.

This call gives drivers the chance to re-initialize the hardware (re-download firmware, etc.). At this point, the driver may assume that the card is in a fresh state and is fully functional. The slot is unfrozen and the driver has full access to PCI config space, memory mapped I/O space and DMA. Interrupts (Legacy, MSI, or MSI-X) will also be available.

Drivers should not restart normal I/O processing operations at this point. If all device drivers report success on this callback, the platform will call resume() to complete the sequence, and let the driver restart normal I/O processing.

A driver can still return a critical failure for this function if it can't get the device operational after reset. If the platform previously tried a soft reset, it might now try a hard reset (power cycle) and then call slot_reset() again. It the device still can't be recovered, there is nothing more that can be done; the platform will typically report a "permanent failure" in such a case. The device will be considered "dead" in this case.

Drivers for multi-function cards will need to coordinate among themselves as to which driver instance will perform any "one-shot" or global device initialization. For example, the Symbios sym53cxx2 driver performs device init only from PCI function 0:

```
+       if (PCI_FUNC(pdev->devfn) == 0)
+               sym_reset_scsi_bus(np, 0);
```

Result codes:

- PCI_ERS_RESULT_DISCONNECT Same as above.

Drivers for PCI Express cards that require a fundamental reset must set the needs_freset bit in the pci_dev structure in their probe function. For example, the QLogic qla2xxx driver sets the needs_freset bit for certain PCI card types:

```
+       /* Set EEH reset type to fundamental if required by hba  */
+       if (IS_QLA24XX(ha) || IS_QLA25XX(ha) || IS_QLA81XX(ha))
+               pdev->needs_freset = 1;
+
```

Platform proceeds either to STEP 5 (Resume Operations) or STEP 6 (Permanent Failure).

Note: The current powerpc implementation does not try a power-cycle reset if the driver returned PCI_ERS_RESULT_DISCONNECT. However, it probably should.

6.1.6 STEP 5: Resume Operations

The platform will call the resume() callback on all affected device drivers if all drivers on the segment have returned PCI_ERS_RESULT_RECOVERED from one of the 3 previous callbacks. The goal of this callback is to tell the driver to restart activity, that everything is back and running. This callback does not return a result code.

At this point, if a new error happens, the platform will restart a new error recovery sequence.

6.1.7 STEP 6: Permanent Failure

A "permanent failure" has occurred, and the platform cannot recover the device. The platform will call error_detected() with a pci_channel_state value of pci_channel_io_perm_failure.

The device driver should, at this point, assume the worst. It should cancel all pending I/O, refuse all new I/O, returning -EIO to higher layers. The device driver should then clean up all of its memory and remove itself from kernel operations, much as it would during system shutdown.

The platform will typically notify the system operator of the permanent failure in some way. If the device is hotplug-capable, the operator will probably want to remove and replace the device. Note, however, not all failures are truly "permanent". Some are caused by over-heating, some by a poorly seated card. Many PCI error events are caused by software bugs, e.g. DMA's to wild addresses or bogus split transactions due to programming errors. See the discussion in powerpc/eeh-pci-error-recovery.txt for additional detail on real-life experience of the causes of software errors.

6.1.8 Conclusion; General Remarks

The way the callbacks are called is platform policy. A platform with no slot reset capability may want to just "ignore" drivers that can't recover (disconnect them) and try to let other cards on the same segment recover. Keep in mind that in most real life cases, though, there will be only one driver per segment.

Now, a note about interrupts. If you get an interrupt and your device is dead or has been isolated, there is a problem :) The current policy is to turn this into a platform policy. That is, the recovery API only requires that:

- There is no guarantee that interrupt delivery can proceed from any device on the segment starting from the error detection and until the slot_reset callback is called, at which point interrupts are expected to be fully operational.
- There is no guarantee that interrupt delivery is stopped, that is, a driver that gets an interrupt after detecting an error, or that detects an error within the interrupt handler such that it prevents proper ack'ing of the interrupt (and thus removal of the source) should just return IRQ_NOTHANDLED. It's up to the platform to deal with that condition, typically by masking the IRQ source during the duration of the error handling. It is expected that the platform "knows" which interrupts are routed to error-management capable slots and can deal with temporarily disabling that IRQ number during error processing (this isn't terribly complex). That means some IRQ latency for other devices sharing the interrupt, but there is simply no other way. High end platforms aren't supposed to share interrupts between many devices anyway :)

Note: Implementation details for the powerpc platform are discussed in the file Documentation/powerpc/eeh-pci-error-recovery.rst

As of this writing, there is a growing list of device drivers with patches implementing error recovery. Not all of these patches are in mainline yet. These may be used as "examples":

- drivers/scsi/ipr
- drivers/scsi/sym53c8xx_2
- drivers/scsi/qla2xxx
- drivers/scsi/lpfc
- drivers/next/bnx2.c
- drivers/next/e100.c
- drivers/net/e1000
- drivers/net/e1000e
- drivers/net/ixgb
- drivers/net/ixgbe
- drivers/net/cxgb3
- drivers/net/s2io.c

6.1.9 The End

CHAPTER

SEVEN

THE PCI EXPRESS ADVANCED ERROR REPORTING DRIVER GUIDE HOWTO

Authors

- T. Long Nguyen <tom.l.nguyen@intel.com>
- Yanmin Zhang <yanmin.zhang@intel.com>

7.1 Overview

7.1.1 About this guide

This guide describes the basics of the PCI Express Advanced Error Reporting (AER) driver and provides information on how to use it, as well as how to enable the drivers of endpoint devices to conform with PCI Express AER driver.

7.1.2 What is the PCI Express AER Driver?

PCI Express error signaling can occur on the PCI Express link itself or on behalf of transactions initiated on the link. PCI Express defines two error reporting paradigms: the baseline capability and the Advanced Error Reporting capability. The baseline capability is required of all PCI Express components providing a minimum defined set of error reporting requirements. Advanced Error Reporting capability is implemented with a PCI Express advanced error reporting extended capability structure providing more robust error reporting.

The PCI Express AER driver provides the infrastructure to support PCI Express Advanced Error Reporting capability. The PCI Express AER driver provides three basic functions:

- Gathers the comprehensive error information if errors occurred.
- Reports error to the users.
- Performs error recovery actions.

AER driver only attaches root ports which support PCI-Express AER capability.

7.2 User Guide

7.2.1 Include the PCI Express AER Root Driver into the Linux Kernel

The PCI Express AER Root driver is a Root Port service driver attached to the PCI Express Port Bus driver. If a user wants to use it, the driver has to be compiled. Option CONFIG_PCIEAER supports this capability. It depends on CONFIG_PCIEPORTBUS, so pls. set CONFIG_PCIEPORTBUS=y and CONFIG_PCIEAER = y.

7.2.2 Load PCI Express AER Root Driver

Some systems have AER support in firmware. Enabling Linux AER support at the same time the firmware handles AER may result in unpredictable behavior. Therefore, Linux does not handle AER events unless the firmware grants AER control to the OS via the ACPI _OSC method. See the PCI FW 3.0 Specification for details regarding _OSC usage.

7.2.3 AER error output

When a PCIe AER error is captured, an error message will be output to console. If it's a correctable error, it is output as a warning. Otherwise, it is printed as an error. So users could choose different log level to filter out correctable error messages.

Below shows an example:

```
0000:50:00.0: PCIe Bus Error: severity=Uncorrected (Fatal),␣
↪type=Transaction Layer, id=0500(Requester ID)
0000:50:00.0:   device [8086:0329] error status/mask=00100000/00000000
0000:50:00.0:    [20] Unsupported Request    (First)
0000:50:00.0:   TLP Header: 04000001 00200a03 05010000 00050100
```

In the example, 'Requester ID' means the ID of the device who sends the error message to root port. Pls. refer to pci express specs for other fields.

7.2.4 AER Statistics / Counters

When PCIe AER errors are captured, the counters / statistics are also exposed in the form of sysfs attributes which are documented at Documentation/ABI/testing/sysfs-bus-pci-devices-aer_stats

7.3 Developer Guide

To enable AER aware support requires a software driver to configure the AER capability structure within its device and to provide callbacks.

To support AER better, developers need understand how AER does work firstly.

PCI Express errors are classified into two types: correctable errors and uncorrectable errors. This classification is based on the impacts of those errors, which may result in degraded performance or function failure.

Correctable errors pose no impacts on the functionality of the interface. The PCI Express protocol can recover without any software intervention or any loss of data. These errors are detected and corrected by hardware. Unlike correctable errors, uncorrectable errors impact functionality of the interface. Uncorrectable errors can cause a particular transaction or a particular PCI Express link to be unreliable. Depending on those error conditions, uncorrectable errors are further classified into non-fatal errors and fatal errors. Non-fatal errors cause the particular transaction to be unreliable, but the PCI Express link itself is fully functional. Fatal errors, on the other hand, cause the link to be unreliable.

When AER is enabled, a PCI Express device will automatically send an error message to the PCIe root port above it when the device captures an error. The Root Port, upon receiving an error reporting message, internally processes and logs the error message in its PCI Express capability structure. Error information being logged includes storing the error reporting agent' s requestor ID into the Error Source Identification Registers and setting the error bits of the Root Error Status Register accordingly. If AER error reporting is enabled in Root Error Command Register, the Root Port generates an interrupt if an error is detected.

Note that the errors as described above are related to the PCI Express hierarchy and links. These errors do not include any device specific errors because device specific errors will still get sent directly to the device driver.

7.3.1 Configure the AER capability structure

AER aware drivers of PCI Express component need change the device control registers to enable AER. They also could change AER registers, including mask and severity registers. Helper function pci_enable_pcie_error_reporting could be used to enable AER. See section 3.3.

7.3.2 Provide callbacks

callback reset_link to reset pci express link

This callback is used to reset the pci express physical link when a fatal error happens. The root port aer service driver provides a default reset_link function, but different upstream ports might have different specifications to reset pci express link, so all upstream ports should provide their own reset_link functions.

Section 3.2.2.2 provides more detailed info on when to call reset_link.

PCI error-recovery callbacks

The PCI Express AER Root driver uses error callbacks to coordinate with downstream device drivers associated with a hierarchy in question when performing error recovery actions.

Data struct pci_driver has a pointer, err_handler, to point to pci_error_handlers who consists of a couple of callback function pointers. AER driver follows the rules defined in pci-error-recovery.txt except pci express specific parts (e.g. reset_link). Pls. refer to pci-error-recovery.txt for detailed definitions of the callbacks.

Below sections specify when to call the error callback functions.

Correctable errors

Correctable errors pose no impacts on the functionality of the interface. The PCI Express protocol can recover without any software intervention or any loss of data. These errors do not require any recovery actions. The AER driver clears the device' s correctable error status register accordingly and logs these errors.

Non-correctable (non-fatal and fatal) errors

If an error message indicates a non-fatal error, performing link reset at upstream is not required. The AER driver calls error_detected(dev, pci_channel_io_normal) to all drivers associated within a hierarchy in question. for example:

```
EndPoint<==>DownstreamPort B<==>UpstreamPort A<==>RootPort
```

If Upstream port A captures an AER error, the hierarchy consists of Downstream port B and EndPoint.

A driver may return PCI_ERS_RESULT_CAN_RECOVER, PCI_ERS_RESULT_DISCONNECT, or PCI_ERS_RESULT_NEED_RESET, depending on whether it can recover or the AER driver calls mmio_enabled as next.

If an error message indicates a fatal error, kernel will broadcast error_detected(dev, pci_channel_io_frozen) to all drivers within a hierarchy in question. Then, performing link reset at upstream is necessary. As different kinds of devices might use different approaches to reset link, AER port service driver is required to provide the function to reset link via callback parameter of pcie_do_recovery() function. If reset_link is not NULL, recovery function will use it to reset the link. If error_detected returns PCI_ERS_RESULT_CAN_RECOVER and reset_link returns PCI_ERS_RESULT_RECOVERED, the error handling goes to mmio_enabled.

7.3.3 helper functions

```
int pci_enable_pcie_error_reporting(struct pci_dev *dev);
```

pci_enable_pcie_error_reporting enables the device to send error messages to root port when an error is detected. Note that devices don't enable the error reporting by default, so device drivers need call this function to enable it.

```
int pci_disable_pcie_error_reporting(struct pci_dev *dev);
```

pci_disable_pcie_error_reporting disables the device to send error messages to root port when an error is detected.

```
int pci_aer_clear_nonfatal_status(struct pci_dev *dev);`
```

pci_aer_clear_nonfatal_status clears non-fatal errors in the uncorrectable error status register.

7.3.4 Frequent Asked Questions

Q: What happens if a PCI Express device driver does not provide an error recovery handler (pci_driver->err_handler is equal to NULL)?

A: The devices attached with the driver won't be recovered. If the error is fatal, kernel will print out warning messages. Please refer to section 3 for more information.

Q: What happens if an upstream port service driver does not provide callback reset_link?

A: Fatal error recovery will fail if the errors are reported by the upstream ports who are attached by the service driver.

Q: How does this infrastructure deal with driver that is not PCI Express aware?

A: This infrastructure calls the error callback functions of the driver when an error happens. But if the driver is not aware of PCI Express, the device might not report its own errors to root port.

Q: What modifications will that driver need to make it compatible with the PCI Express AER Root driver?

A: It could call the helper functions to enable AER in devices and cleanup uncorrectable status register. Pls. refer to section 3.3.

7.4 Software error injection

Debugging PCIe AER error recovery code is quite difficult because it is hard to trigger real hardware errors. Software based error injection can be used to fake various kinds of PCIe errors.

First you should enable PCIe AER software error injection in kernel configuration, that is, following item should be in your .config.

CONFIG_PCIEAER_INJECT=y or CONFIG_PCIEAER_INJECT=m

After reboot with new kernel or insert the module, a device file named /dev/aer_inject should be created.

Then, you need a user space tool named aer-inject, which can be gotten from:

https://git.kernel.org/cgit/linux/kernel/git/gong.chen/aer-inject.git/

More information about aer-inject can be found in the document comes with its source code.

CHAPTER

EIGHT

PCI ENDPOINT FRAMEWORK

This document is a guide to use the PCI Endpoint Framework in order to create endpoint controller driver, endpoint function driver, and using configfs interface to bind the function driver to the controller driver.

8.1 Introduction

Linux has a comprehensive PCI subsystem to support PCI controllers that operates in Root Complex mode. The subsystem has capability to scan PCI bus, assign memory resources and IRQ resources, load PCI driver (based on vendor ID, device ID), support other services like hot-plug, power management, advanced error reporting and virtual channels.

However the PCI controller IP integrated in some SoCs is capable of operating either in Root Complex mode or Endpoint mode. PCI Endpoint Framework will add endpoint mode support in Linux. This will help to run Linux in an EP system which can have a wide variety of use cases from testing or validation, co-processor accelerator, etc.

8.2 PCI Endpoint Core

The PCI Endpoint Core layer comprises 3 components: the Endpoint Controller library, the Endpoint Function library, and the configfs layer to bind the endpoint function with the endpoint controller.

8.2.1 PCI Endpoint Controller(EPC) Library

The EPC library provides APIs to be used by the controller that can operate in endpoint mode. It also provides APIs to be used by function driver/library in order to implement a particular endpoint function.

APIs for the PCI controller Driver

This section lists the APIs that the PCI Endpoint core provides to be used by the PCI controller driver.

- devm_pci_epc_create()/pci_epc_create()

  The PCI controller driver should implement the following ops:

  - write_header: ops to populate configuration space header
  - set_bar: ops to configure the BAR
  - clear_bar: ops to reset the BAR
  - alloc_addr_space: ops to allocate in PCI controller address space
  - free_addr_space: ops to free the allocated address space
  - raise_irq: ops to raise a legacy, MSI or MSI-X interrupt
  - start: ops to start the PCI link
  - stop: ops to stop the PCI link

  The PCI controller driver can then create a new EPC device by invoking devm_pci_epc_create()/pci_epc_create().

- devm_pci_epc_destroy()/pci_epc_destroy()

  The PCI controller driver can destroy the EPC device created by either devm_pci_epc_create() or pci_epc_create() using devm_pci_epc_destroy() or pci_epc_destroy().

- pci_epc_linkup()

  In order to notify all the function devices that the EPC device to which they are linked has established a link with the host, the PCI controller driver should invoke pci_epc_linkup().

- pci_epc_mem_init()

  Initialize the pci_epc_mem structure used for allocating EPC addr space.

- pci_epc_mem_exit()

  Cleanup the pci_epc_mem structure allocated during pci_epc_mem_init().

EPC APIs for the PCI Endpoint Function Driver

This section lists the APIs that the PCI Endpoint core provides to be used by the PCI endpoint function driver.

- pci_epc_write_header()

  The PCI endpoint function driver should use pci_epc_write_header() to write the standard configuration header to the endpoint controller.

- pci_epc_set_bar()

The PCI endpoint function driver should use pci_epc_set_bar() to configure the Base Address Register in order for the host to assign PCI addr space. Register space of the function driver is usually configured using this API.

- pci_epc_clear_bar()

  The PCI endpoint function driver should use pci_epc_clear_bar() to reset the BAR.

- pci_epc_raise_irq()

  The PCI endpoint function driver should use pci_epc_raise_irq() to raise Legacy Interrupt, MSI or MSI-X Interrupt.

- pci_epc_mem_alloc_addr()

  The PCI endpoint function driver should use pci_epc_mem_alloc_addr(), to allocate memory address from EPC addr space which is required to access RC's buffer

- pci_epc_mem_free_addr()

  The PCI endpoint function driver should use pci_epc_mem_free_addr() to free the memory space allocated using pci_epc_mem_alloc_addr().

Other EPC APIs

There are other APIs provided by the EPC library. These are used for binding the EPF device with EPC device. pci-ep-cfs.c can be used as reference for using these APIs.

- pci_epc_get()

  Get a reference to the PCI endpoint controller based on the device name of the controller.

- pci_epc_put()

  Release the reference to the PCI endpoint controller obtained using pci_epc_get()

- pci_epc_add_epf()

  Add a PCI endpoint function to a PCI endpoint controller. A PCIe device can have up to 8 functions according to the specification.

- pci_epc_remove_epf()

  Remove the PCI endpoint function from PCI endpoint controller.

- pci_epc_start()

  The PCI endpoint function driver should invoke pci_epc_start() once it has configured the endpoint function and wants to start the PCI link.

- pci_epc_stop()

The PCI endpoint function driver should invoke pci_epc_stop() to stop the PCI LINK.

8.2.2 PCI Endpoint Function(EPF) Library

The EPF library provides APIs to be used by the function driver and the EPC library to provide endpoint mode functionality.

EPF APIs for the PCI Endpoint Function Driver

This section lists the APIs that the PCI Endpoint core provides to be used by the PCI endpoint function driver.

- pci_epf_register_driver()

  The PCI Endpoint Function driver should implement the following ops:

  - bind: ops to perform when a EPC device has been bound to EPF device
  - unbind: ops to perform when a binding has been lost between a EPC device and EPF device
  - linkup: ops to perform when the EPC device has established a connection with a host system

  The PCI Function driver can then register the PCI EPF driver by using pci_epf_register_driver().

- pci_epf_unregister_driver()

  The PCI Function driver can unregister the PCI EPF driver by using pci_epf_unregister_driver().

- pci_epf_alloc_space()

  The PCI Function driver can allocate space for a particular BAR using pci_epf_alloc_space().

- pci_epf_free_space()

  The PCI Function driver can free the allocated space (using pci_epf_alloc_space) by invoking pci_epf_free_space().

APIs for the PCI Endpoint Controller Library

This section lists the APIs that the PCI Endpoint core provides to be used by the PCI endpoint controller library.

- pci_epf_linkup()

  The PCI endpoint controller library invokes pci_epf_linkup() when the EPC device has established the connection to the host.

Other EPF APIs

There are other APIs provided by the EPF library. These are used to notify the function driver when the EPF device is bound to the EPC device. pci-ep-cfs.c can be used as reference for using these APIs.

- pci_epf_create()

  Create a new PCI EPF device by passing the name of the PCI EPF device. This name will be used to bind the the EPF device to a EPF driver.

- pci_epf_destroy()

  Destroy the created PCI EPF device.

- pci_epf_bind()

  pci_epf_bind() should be invoked when the EPF device has been bound to a EPC device.

- pci_epf_unbind()

  pci_epf_unbind() should be invoked when the binding between EPC device and EPF device is lost.

8.3 Configuring PCI Endpoint Using CONFIGFS

Author Kishon Vijay Abraham I <kishon@ti.com>

The PCI Endpoint Core exposes configfs entry (pci_ep) to configure the PCI endpoint function and to bind the endpoint function with the endpoint controller. (For introducing other mechanisms to configure the PCI Endpoint Function refer to [1]).

8.3.1 Mounting configfs

The PCI Endpoint Core layer creates pci_ep directory in the mounted configfs directory. configfs can be mounted using the following command:

```
mount -t configfs none /sys/kernel/config
```

8.3.2 Directory Structure

The pci_ep configfs has two directories at its root: controllers and functions. Every EPC device present in the system will have an entry in the controllers directory and and every EPF driver present in the system will have an entry in the functions directory.

```
/sys/kernel/config/pci_ep/
        .. controllers/
        .. functions/
```

8.3.3 Creating EPF Device

Every registered EPF driver will be listed in controllers directory. The entries corresponding to EPF driver will be created by the EPF core.

```
/sys/kernel/config/pci_ep/functions/
        .. <EPF Driver1>/
                ... <EPF Device 11>/
                ... <EPF Device 21>/
        .. <EPF Driver2>/
                ... <EPF Device 12>/
                ... <EPF Device 22>/
```

In order to create a <EPF device> of the type probed by <EPF Driver>, the user has to create a directory inside <EPF DriverN>.

Every <EPF device> directory consists of the following entries that can be used to configure the standard configuration header of the endpoint function. (These entries are created by the framework when any new <EPF Device> is created)

```
.. <EPF Driver1>/
        ... <EPF Device 11>/
                ... vendorid
                ... deviceid
                ... revid
                ... progif_code
                ... subclass_code
                ... baseclass_code
                ... cache_line_size
                ... subsys_vendor_id
                ... subsys_id
                ... interrupt_pin
```

8.3.4 EPC Device

Every registered EPC device will be listed in controllers directory. The entries corresponding to EPC device will be created by the EPC core.

```
/sys/kernel/config/pci_ep/controllers/
        .. <EPC Device1>/
                ... <Symlink EPF Device11>/
                ... <Symlink EPF Device12>/
                ... start
        .. <EPC Device2>/
                ... <Symlink EPF Device21>/
                ... <Symlink EPF Device22>/
                ... start
```

The <EPC Device> directory will have a list of symbolic links to <EPF Device>. These symbolic links should be created by the user to represent the functions present in the endpoint device.

The <EPC Device> directory will also have a start field. Once "1" is written to this field, the endpoint device will be ready to establish the link with the host. This is usually done after all the EPF devices are created and linked with the EPC device.

```
| controllers/
       | <Directory: EPC name>/
                | <Symbolic Link: Function>
                | start
| functions/
       | <Directory: EPF driver>/
                | <Directory: EPF device>/
                        | vendorid
                        | deviceid
                        | revid
                        | progif_code
                        | subclass_code
                        | baseclass_code
                        | cache_line_size
                        | subsys_vendor_id
                        | subsys_id
                        | interrupt_pin
                        | function
```

[1] Introduction

8.4 PCI Test Function

Author Kishon Vijay Abraham I <kishon@ti.com>

Traditionally PCI RC has always been validated by using standard PCI cards like ethernet PCI cards or USB PCI cards or SATA PCI cards. However with the addition of EP-core in linux kernel, it is possible to configure a PCI controller that can operate in EP mode to work as a test device.

The PCI endpoint test device is a virtual device (defined in software) used to test the endpoint functionality and serve as a sample driver for other PCI endpoint devices (to use the EP framework).

The PCI endpoint test device has the following registers:

1) PCI_ENDPOINT_TEST_MAGIC
2) PCI_ENDPOINT_TEST_COMMAND
3) PCI_ENDPOINT_TEST_STATUS
4) PCI_ENDPOINT_TEST_SRC_ADDR
5) PCI_ENDPOINT_TEST_DST_ADDR
6) PCI_ENDPOINT_TEST_SIZE
7) PCI_ENDPOINT_TEST_CHECKSUM
8) PCI_ENDPOINT_TEST_IRQ_TYPE
9) PCI_ENDPOINT_TEST_IRQ_NUMBER

- PCI_ENDPOINT_TEST_MAGIC

This register will be used to test BAR0. A known pattern will be written and read back from MAGIC register to verify BAR0.

- PCI_ENDPOINT_TEST_COMMAND

This register will be used by the host driver to indicate the function that the endpoint device must perform.

Bitfield	Description
Bit 0	raise legacy IRQ
Bit 1	raise MSI IRQ
Bit 2	raise MSI-X IRQ
Bit 3	read command (read data from RC buffer)
Bit 4	write command (write data to RC buffer)
Bit 5	copy command (copy data from one RC buffer to another RC buffer)

- PCI_ENDPOINT_TEST_STATUS

This register reflects the status of the PCI endpoint device.

Bitfield	Description
Bit 0	read success
Bit 1	read fail
Bit 2	write success
Bit 3	write fail
Bit 4	copy success
Bit 5	copy fail
Bit 6	IRQ raised
Bit 7	source address is invalid
Bit 8	destination address is invalid

- PCI_ENDPOINT_TEST_SRC_ADDR

This register contains the source address (RC buffer address) for the COPY/READ command.

- PCI_ENDPOINT_TEST_DST_ADDR

This register contains the destination address (RC buffer address) for the COPY/WRITE command.

- PCI_ENDPOINT_TEST_IRQ_TYPE

This register contains the interrupt type (Legacy/MSI) triggered for the READ/WRITE/COPY and raise IRQ (Legacy/MSI) commands.

Possible types:

Legacy	0
MSI	1
MSI-X	2

- PCI_ENDPOINT_TEST_IRQ_NUMBER

This register contains the triggered ID interrupt.

Admissible values:

Legacy	0
MSI	[1 .. 32]
MSI-X	[1 .. 2048]

8.5 PCI Test User Guide

Author Kishon Vijay Abraham I <kishon@ti.com>

This document is a guide to help users use pci-epf-test function driver and pci_endpoint_test host driver for testing PCI. The list of steps to be followed in the host side and EP side is given below.

8.5.1 Endpoint Device

Endpoint Controller Devices

To find the list of endpoint controller devices in the system:

```
# ls /sys/class/pci_epc/
  51000000.pcie_ep
```

If PCI_ENDPOINT_CONFIGFS is enabled:

```
# ls /sys/kernel/config/pci_ep/controllers
  51000000.pcie_ep
```

Endpoint Function Drivers

To find the list of endpoint function drivers in the system:

```
# ls /sys/bus/pci-epf/drivers
  pci_epf_test
```

If PCI_ENDPOINT_CONFIGFS is enabled:

```
# ls /sys/kernel/config/pci_ep/functions
  pci_epf_test
```

Creating pci-epf-test Device

PCI endpoint function device can be created using the configfs. To create pci-epf-test device, the following commands can be used:

```
# mount -t configfs none /sys/kernel/config
# cd /sys/kernel/config/pci_ep/
# mkdir functions/pci_epf_test/func1
```

The “mkdir func1”above creates the pci-epf-test function device that will be probed by pci_epf_test driver.

The PCI endpoint framework populates the directory with the following configurable fields:

```
# ls functions/pci_epf_test/func1
  baseclass_code        interrupt_pin   progif_code     subsys_id
  cache_line_size       msi_interrupts  revid           subsys_vendorid
  deviceid              msix_interrupts subclass_code   vendorid
```

The PCI endpoint function driver populates these entries with default values when the device is bound to the driver. The pci-epf-test driver populates vendorid with 0xffff and interrupt_pin with 0x0001:

```
# cat functions/pci_epf_test/func1/vendorid
  0xffff
# cat functions/pci_epf_test/func1/interrupt_pin
  0x0001
```

Configuring pci-epf-test Device

The user can configure the pci-epf-test device using configfs entry. In order to change the vendorid and the number of MSI interrupts used by the function device, the following commands can be used:

```
# echo 0x104c > functions/pci_epf_test/func1/vendorid
# echo 0xb500 > functions/pci_epf_test/func1/deviceid
# echo 16 > functions/pci_epf_test/func1/msi_interrupts
# echo 8 > functions/pci_epf_test/func1/msix_interrupts
```

Binding pci-epf-test Device to EP Controller

In order for the endpoint function device to be useful, it has to be bound to a PCI endpoint controller driver. Use the configfs to bind the function device to one of the controller driver present in the system:

```
# ln -s functions/pci_epf_test/func1 controllers/51000000.pcie_ep/
```

Once the above step is completed, the PCI endpoint is ready to establish a link with the host.

Start the Link

In order for the endpoint device to establish a link with the host, the _start_ field should be populated with ‘1’:

```
# echo 1 > controllers/51000000.pcie_ep/start
```

8.5.2 RootComplex Device

lspci Output

Note that the devices listed here correspond to the value populated in 1.4 above:

```
00:00.0 PCI bridge: Texas Instruments Device 8888 (rev 01)
01:00.0 Unassigned class [ff00]: Texas Instruments Device b500
```

Using Endpoint Test function Device

pcitest.sh added in tools/pci/ can be used to run all the default PCI endpoint tests. To compile this tool the following commands should be used:

```
# cd <kernel-dir>
# make -C tools/pci
```

or if you desire to compile and install in your system:

```
# cd <kernel-dir>
# make -C tools/pci install
```

The tool and script will be located in <rootfs>/usr/bin/

pcitest.sh Output

```
# pcitest.sh
BAR tests

BAR0:           OKAY
BAR1:           OKAY
BAR2:           OKAY
BAR3:           OKAY
BAR4:           NOT OKAY
BAR5:           NOT OKAY

Interrupt tests

SET IRQ TYPE TO LEGACY:         OKAY
LEGACY IRQ:     NOT OKAY
SET IRQ TYPE TO MSI:            OKAY
MSI1:           OKAY
MSI2:           OKAY
MSI3:           OKAY
MSI4:           OKAY
MSI5:           OKAY
MSI6:           OKAY
MSI7:           OKAY
MSI8:           OKAY
MSI9:           OKAY
MSI10:          OKAY
MSI11:          OKAY
MSI12:          OKAY
```

(continues on next page)

(continued from previous page)

```
MSI13:          OKAY
MSI14:          OKAY
MSI15:          OKAY
MSI16:          OKAY
MSI17:          NOT OKAY
MSI18:          NOT OKAY
MSI19:          NOT OKAY
MSI20:          NOT OKAY
MSI21:          NOT OKAY
MSI22:          NOT OKAY
MSI23:          NOT OKAY
MSI24:          NOT OKAY
MSI25:          NOT OKAY
MSI26:          NOT OKAY
MSI27:          NOT OKAY
MSI28:          NOT OKAY
MSI29:          NOT OKAY
MSI30:          NOT OKAY
MSI31:          NOT OKAY
MSI32:          NOT OKAY
SET IRQ TYPE TO MSI-X:          OKAY
MSI-X1:         OKAY
MSI-X2:         OKAY
MSI-X3:         OKAY
MSI-X4:         OKAY
MSI-X5:         OKAY
MSI-X6:         OKAY
MSI-X7:         OKAY
MSI-X8:         OKAY
MSI-X9:         NOT OKAY
MSI-X10:        NOT OKAY
MSI-X11:        NOT OKAY
MSI-X12:        NOT OKAY
MSI-X13:        NOT OKAY
MSI-X14:        NOT OKAY
MSI-X15:        NOT OKAY
MSI-X16:        NOT OKAY
[...]
MSI-X2047:      NOT OKAY
MSI-X2048:      NOT OKAY

Read Tests

SET IRQ TYPE TO MSI:            OKAY
READ (      1 bytes):           OKAY
READ (   1024 bytes):           OKAY
READ (   1025 bytes):           OKAY
READ (1024000 bytes):           OKAY
READ (1024001 bytes):           OKAY

Write Tests

WRITE (      1 bytes):          OKAY
WRITE (   1024 bytes):          OKAY
WRITE (   1025 bytes):          OKAY
WRITE (1024000 bytes):          OKAY
```

(continues on next page)

(continued from previous page)

```
WRITE (1024001 bytes):          OKAY

Copy Tests

COPY (      1 bytes):           OKAY
COPY (   1024 bytes):           OKAY
COPY (   1025 bytes):           OKAY
COPY (1024000 bytes):           OKAY
COPY (1024001 bytes):           OKAY
```

CHAPTER

NINE

BOOT INTERRUPTS

Author

- Sean V Kelley <sean.v.kelley@linux.intel.com>

9.1 Overview

On PCI Express, interrupts are represented with either MSI or inbound interrupt messages (Assert_INTx/Deassert_INTx). The integrated IO-APIC in a given Core IO converts the legacy interrupt messages from PCI Express to MSI interrupts. If the IO-APIC is disabled (via the mask bits in the IO-APIC table entries), the messages are routed to the legacy PCH. This in-band interrupt mechanism was traditionally necessary for systems that did not support the IO-APIC and for boot. Intel in the past has used the term "boot interrupts" to describe this mechanism. Further, the PCI Express protocol describes this in-band legacy wire-interrupt INTx mechanism for I/O devices to signal PCI-style level interrupts. The subsequent paragraphs describe problems with the Core IO handling of INTx message routing to the PCH and mitigation within BIOS and the OS.

9.2 Issue

When in-band legacy INTx messages are forwarded to the PCH, they in turn trigger a new interrupt for which the OS likely lacks a handler. When an interrupt goes unhandled over time, they are tracked by the Linux kernel as Spurious Interrupts. The IRQ will be disabled by the Linux kernel after it reaches a specific count with the error "nobody cared". This disabled IRQ now prevents valid usage by an existing interrupt which may happen to share the IRQ line:

```
irq 19: nobody cared (try booting with the "irqpoll" option)
CPU: 0 PID: 2988 Comm: irq/34-nipalk Tainted: 4.14.87-rt49-02410-g4a640ec-
↪dirty #1
Hardware name: National Instruments NI PXIe-8880/NI PXIe-8880, BIOS 2.1.
↪5f1 01/09/2020
Call Trace:

<IRQ>
 ? dump_stack+0x46/0x5e
 ? __report_bad_irq+0x2e/0xb0
 ? note_interrupt+0x242/0x290
```

(continues on next page)

(continued from previous page)

```
 ? nNIKAL100_memoryRead16+0x8/0x10 [nikal]
 ? handle_irq_event_percpu+0x55/0x70
 ? handle_irq_event+0x4f/0x80
 ? handle_fasteoi_irq+0x81/0x180
 ? handle_irq+0x1c/0x30
 ? do_IRQ+0x41/0xd0
 ? common_interrupt+0x84/0x84
</IRQ>

handlers:
irq_default_primary_handler threaded usb_hcd_irq
Disabling IRQ #19
```

9.3 Conditions

The use of threaded interrupts is the most likely condition to trigger this problem today. Threaded interrupts may not be reenabled after the IRQ handler wakes. These “one shot” conditions mean that the threaded interrupt needs to keep the interrupt line masked until the threaded handler has run. Especially when dealing with high data rate interrupts, the thread needs to run to completion; otherwise some handlers will end up in stack overflows since the interrupt of the issuing device is still active.

9.4 Affected Chipsets

The legacy interrupt forwarding mechanism exists today in a number of devices including but not limited to chipsets from AMD/ATI, Broadcom, and Intel. Changes made through the mitigations below have been applied to drivers/pci/quirks.c

Starting with ICX there are no longer any IO-APICs in the Core IO’s devices. IO-APIC is only in the PCH. Devices connected to the Core IO’s PCIe Root Ports will use native MSI/MSI-X mechanisms.

9.5 Mitigations

The mitigations take the form of PCI quirks. The preference has been to first identify and make use of a means to disable the routing to the PCH. In such a case a quirk to disable boot interrupt generation can be added.[1]

Intel® 6300ESB I/O Controller Hub

Alternate Base Address Register: BIE: Boot Interrupt Enable

0	Boot interrupt is enabled.
1	Boot interrupt is disabled.

[1] https://lore.kernel.org/r/12131949181903-git-send-email-sassmann@suse.de/

Intel® Sandy Bridge through Sky Lake based Xeon servers:

Coherent Interface Protocol Interrupt Control

dis_intx_route2pch/dis_intx_route2ich/dis_intx_route2dmi2: When this bit is set. Local INTx messages received from the Intel® Quick Data DMA/PCI Express ports are not routed to legacy PCH - they are either converted into MSI via the integrated IO-APIC (if the IO-APIC mask bit is clear in the appropriate entries) or cause no further action (when mask bit is set)

In the absence of a way to directly disable the routing, another approach has been to make use of PCI Interrupt pin to INTx routing tables for purposes of redirecting the interrupt handler to the rerouted interrupt line by default. Therefore, on chipsets where this INTx routing cannot be disabled, the Linux kernel will reroute the valid interrupt to its legacy interrupt. This redirection of the handler will prevent the occurrence of the spurious interrupt detection which would ordinarily disable the IRQ line due to excessive unhandled counts.[2]

The config option X86_REROUTE_FOR_BROKEN_BOOT_IRQS exists to enable (or disable) the redirection of the interrupt handler to the PCH interrupt line. The option can be overridden by either pci=ioapicreroute or pci=noioapicreroute.[3]

9.6 More Documentation

There is an overview of the legacy interrupt handling in several datasheets (6300ESB and 6700PXH below). While largely the same, it provides insight into the evolution of its handling with chipsets.

9.6.1 Example of disabling of the boot interrupt

- Intel® 6300ESB I/O Controller Hub (Document # 300641-004US) 5.7.3 Boot Interrupt https://www.intel.com/content/dam/doc/datasheet/6300esb-io-controller-hub-datasheet.pdf
- Intel® Xeon® Processor E5-1600/2400/2600/4600 v3 Product Families Datasheet - Volume 2: Registers (Document # 330784-003) 6.6.41 cipintrc Coherent Interface Protocol Interrupt Control https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v3-datasheet-vol-2.pdf

[2] https://lore.kernel.org/r/12131949182094-git-send-email-sassmann@suse.de/
[3] https://lore.kernel.org/r/487C8EA7.6020205@suse.de/

9.6.2 Example of handler rerouting

- Intel® 6700PXH 64-bit PCI Hub (Document # 302628) 2.15.2 PCI Express Legacy INTx Support and Boot Interrupt https://www.intel.com/content/dam/doc/datasheet/6700pxh-64-bit-pci-hub-datasheet.pdf

If you have any legacy PCI interrupt questions that aren’t answered, email me.

Cheers, Sean V Kelley sean.v.kelley@linux.intel.com