Linux I2c Documentation

The kernel development community

Jul 14, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Introduction to I2C and SMBus

I^{2}C (pronounce: I squared C and written I2C in the kernel documentation) is a protocol developed by Philips. It is a slow two-wire protocol (variable speed, up to 400 kHz), with a high speed extension (3.4 MHz). It provides an inexpensive bus for connecting many types of devices with infrequent or low bandwidth communications needs. I2C is widely used with embedded systems. Some systems use variants that don't meet branding requirements, and so are not advertised as being I2C but come under different names, e.g. TWI (Two Wire Interface), IIC.

The official I2C specification is the "I2C-bus specification and user manual" (UM10204) published by NXP Semiconductors.

SMBus (System Management Bus) is based on the I2C protocol, and is mostly a subset of I2C protocols and signaling. Many I2C devices will work on an SMBus, but some SMBus protocols add semantics beyond what is required to achieve I2C branding. Modern PC mainboards rely on SMBus. The most common devices connected through SMBus are RAM modules configured using I2C EEPROMs, and hardware monitoring chips.

Because the SMBus is mostly a subset of the generalized I2C bus, we can use its protocols on many I2C systems. However, there are systems that don't meet both SMBus and I2C electrical constraints; and others which can't implement all the common SMBus protocol semantics or messages.

1.1.1 Terminology

Using the terminology from the official documentation, the I2C bus connects one or more master chips and one or more slave chips.

A **master** chip is a node that starts communications with slaves. In the Linux kernel implementation it is called an **adapter** or bus. Adapter drivers are in the `drivers/i2c/busses/` subdirectory.

An **algorithm** contains general code that can be used to implement a whole class of I2C adapters. Each specific adapter driver either depends on an algorithm driver in the `drivers/i2c/algos/` subdirectory, or includes its own implementation.

A **slave** chip is a node that responds to communications when addressed by the master. In Linux it is called a **client**. Client drivers are kept in a directory specific

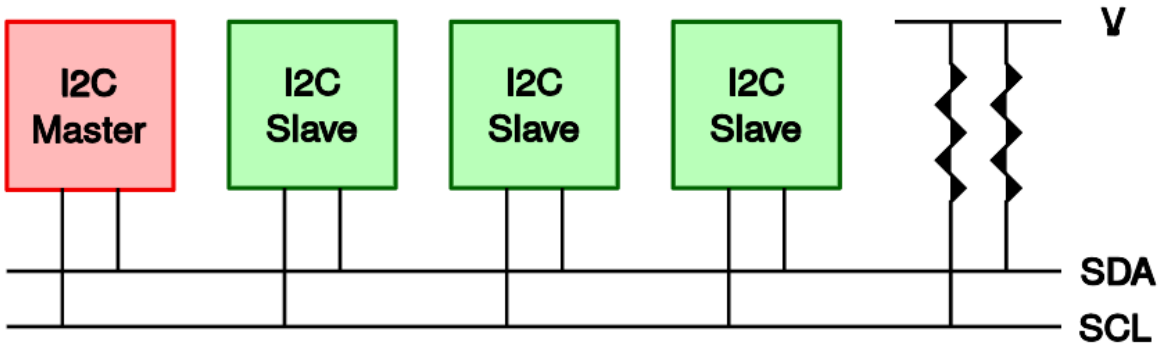


Fig. 1: Simple I2C bus

to the feature they provide, for example `drivers/media/gpio/` for GPIO expanders and `drivers/media/i2c/` for video-related chips.

For the example configuration in figure, you will need a driver for your I2C adapter, and drivers for your I2C devices (usually one driver for each device).

1.2 The I2C Protocol

This document describes the I2C protocol. Or will, when it is finished :-)

1.2.1 Key to symbols

S	Start condition
P	Stop condition
Rd/Wr (1 bit)	Read/Write bit. Rd equals 1, Wr equals 0.
A, NA (1 bit)	Acknowledge (ACK) and Not Acknowledge (NACK) bit
Addr (7 bits)	I2C 7 bit address. Note that this can be expanded as usual to get a 10 bit I2C address.
Comm (8 bits)	Command byte, a data byte which often selects a register on the device.
Data (8 bits)	A plain data byte. Sometimes, I write DataLow, DataHigh for 16 bit data.
Count (8 bits)	A data byte containing the length of a block operation.
[..]	Data sent by I2C device, as opposed to data sent by the host adapter.

1.2.2 Simple send transaction

Implemented by i2c_master_send():

```
S Addr Wr [A] Data [A] Data [A] ... [A] Data [A] P
```

1.2.3 Simple receive transaction

Implemented by i2c_master_recv():

```
S Addr Rd [A] [Data] A [Data] A ... A [Data] NA P
```

1.2.4 Combined transactions

Implemented by i2c_transfer().

They are just like the above transactions, but instead of a stop condition P a start condition S is sent and the transaction continues. An example of a byte read, followed by a byte write:

```
S Addr Rd [A] [Data] NA S Addr Wr [A] Data [A] P
```

1.2.5 Modified transactions

The following modifications to the I2C protocol can also be generated by setting these flags for I2C messages. With the exception of I2C_M_NOSTART, they are usually only needed to work around device issues:

I2C_M_IGNORE_NAK: Normally message is interrupted immediately if there is [NA] from the client. Setting this flag treats any [NA] as [A], and all of message is sent. These messages may still fail to SCL lo->hi timeout.

I2C_M_NO_RD_ACK: In a read message, master A/NA bit is skipped.

I2C_M_NOSTART: In a combined transaction, no 'S Addr Wr/Rd [A]' is generated at some point. For example, setting I2C_M_NOSTART on the second partial message generates something like:

```
S Addr Rd [A] [Data] NA Data [A] P
```

If you set the I2C_M_NOSTART variable for the first partial message, we do not generate Addr, but we do generate the start condition S. This will probably confuse all other clients on your bus, so don't try this.

This is often used to gather transmits from multiple data buffers in system memory into something that appears as a single transfer to the I2C device but may also be used between direction changes by some rare devices.

I2C_M_REV_DIR_ADDR: This toggles the Rd/Wr flag. That is, if you want to do a write, but need to emit an Rd instead of a Wr, or vice versa, you set this flag. For example:

```
S Addr Rd [A] Data [A] Data [A] ... [A] Data [A] P
```

I2C_M_STOP: Force a stop condition (P) after the message. Some I2C related protocols like SCCB require that. Normally, you really don't want to get interrupted between the messages of one transfer.

1.3 The SMBus Protocol

The following is a summary of the SMBus protocol. It applies to all revisions of the protocol (1.0, 1.1, and 2.0). Certain protocol features which are not supported by this package are briefly described at the end of this document.

Some adapters understand only the SMBus (System Management Bus) protocol, which is a subset from the I2C protocol. Fortunately, many devices use only the same subset, which makes it possible to put them on an SMBus.

If you write a driver for some I2C device, please try to use the SMBus commands if at all possible (if the device uses only that subset of the I2C protocol). This makes it possible to use the device driver on both SMBus adapters and I2C adapters (the SMBus command set is automatically translated to I2C on I2C adapters, but plain I2C commands can not be handled at all on most pure SMBus adapters).

Below is a list of SMBus protocol operations, and the functions executing them. Note that the names used in the SMBus protocol specifications usually don't match these function names. For some of the operations which pass a single data byte, the functions using SMBus protocol operation names execute a different protocol operation entirely.

Each transaction type corresponds to a functionality flag. Before calling a transaction function, a device driver should always check (just once) for the corresponding functionality flag to ensure that the underlying I2C adapter supports the transaction in question. See I2C/SMBus Functionality for the details.

1.3.1 Key to symbols

S	Start condition
P	Stop condition
Rd/Wr (1 bit)	Read/Write bit. Rd equals 1, Wr equals 0.
A, NA (1 bit)	Acknowledge (ACK) and Not Acknowledge (NACK) bit
Addr (7 bits)	I2C 7 bit address. Note that this can be expanded as usual to get a 10 bit I2C address.
Comm (8 bits)	Command byte, a data byte which often selects a register on the device.
Data (8 bits)	A plain data byte. Sometimes, I write DataLow, DataHigh for 16 bit data.
Count (8 bits)	A data byte containing the length of a block operation.
[..]	Data sent by I2C device, as opposed to data sent by the host adapter.

1.3.2 SMBus Quick Command

This sends a single bit to the device, at the place of the Rd/Wr bit:

```
S Addr Rd/Wr [A] P
```

Functionality flag: I2C_FUNC_SMBUS_QUICK

1.3.3 SMBus Receive Byte

Implemented by i2c_smbus_read_byte()

This reads a single byte from a device, without specifying a device register. Some devices are so simple that this interface is enough; for others, it is a shorthand if you want to read the same register as in the previous SMBus command:

```
S Addr Rd [A] [Data] NA P
```

Functionality flag: I2C_FUNC_SMBUS_READ_BYTE

1.3.4 SMBus Send Byte

Implemented by i2c_smbus_write_byte()

This operation is the reverse of Receive Byte: it sends a single byte to a device. See Receive Byte for more information.

```
S Addr Wr [A] Data [A] P
```

Functionality flag: I2C_FUNC_SMBUS_WRITE_BYTE

1.3.5 SMBus Read Byte

Implemented by i2c_smbus_read_byte_data()

This reads a single byte from a device, from a designated register. The register is specified through the Comm byte:

```
S Addr Wr [A] Comm [A] S Addr Rd [A] [Data] NA P
```

Functionality flag: I2C_FUNC_SMBUS_READ_BYTE_DATA

1.3.6 SMBus Read Word

Implemented by i2c_smbus_read_word_data()

This operation is very like Read Byte; again, data is read from a device, from a designated register that is specified through the Comm byte. But this time, the data is a complete word (16 bits):

```
S Addr Wr [A] Comm [A] S Addr Rd [A] [DataLow] A [DataHigh] NA P
```

Functionality flag: I2C_FUNC_SMBUS_READ_WORD_DATA

Note the convenience function i2c_smbus_read_word_swapped() is available for reads where the two data bytes are the other way around (not SMBus compliant, but very popular.)

1.3.7 SMBus Write Byte

Implemented by i2c_smbus_write_byte_data()

This writes a single byte to a device, to a designated register. The register is specified through the Comm byte. This is the opposite of the Read Byte operation.

```
S Addr Wr [A] Comm [A] Data [A] P
```

Functionality flag: I2C_FUNC_SMBUS_WRITE_BYTE_DATA

1.3.8 SMBus Write Word

Implemented by i2c_smbus_write_word_data()

This is the opposite of the Read Word operation. 16 bits of data are written to a device, to the designated register that is specified through the Comm byte:

```
S Addr Wr [A] Comm [A] DataLow [A] DataHigh [A] P
```

Functionality flag: I2C_FUNC_SMBUS_WRITE_WORD_DATA

Note the convenience function i2c_smbus_write_word_swapped() is available for writes where the two data bytes are the other way around (not SMBus compliant, but very popular.)

1.3.9 SMBus Process Call

This command selects a device register (through the Comm byte), sends 16 bits of data to it, and reads 16 bits of data in return:

```
S Addr Wr [A] Comm [A] DataLow [A] DataHigh [A]
                               S Addr Rd [A] [DataLow] A [DataHigh] NA P
```

Functionality flag: I2C_FUNC_SMBUS_PROC_CALL

1.3.10 SMBus Block Read

Implemented by i2c_smbus_read_block_data()

This command reads a block of up to 32 bytes from a device, from a designated register that is specified through the Comm byte. The amount of data is specified by the device in the Count byte.

```
S Addr Wr [A] Comm [A]
           S Addr Rd [A] [Count] A [Data] A [Data] A ... A [Data] NA P
```

Functionality flag: I2C_FUNC_SMBUS_READ_BLOCK_DATA

1.3.11 SMBus Block Write

Implemented by i2c_smbus_write_block_data()

The opposite of the Block Read command, this writes up to 32 bytes to a device, to a designated register that is specified through the Comm byte. The amount of data is specified in the Count byte.

```
S Addr Wr [A] Comm [A] Count [A] Data [A] Data [A] ... [A] Data [A] P
```

Functionality flag: I2C_FUNC_SMBUS_WRITE_BLOCK_DATA

1.3.12 SMBus Block Write - Block Read Process Call

SMBus Block Write - Block Read Process Call was introduced in Revision 2.0 of the specification.

This command selects a device register (through the Comm byte), sends 1 to 31 bytes of data to it, and reads 1 to 31 bytes of data in return:

```
S Addr Wr [A] Comm [A] Count [A] Data [A] ...
                               S Addr Rd [A] [Count] A [Data] ... A P
```

Functionality flag: I2C_FUNC_SMBUS_BLOCK_PROC_CALL

1.3.13 SMBus Host Notify

This command is sent from a SMBus device acting as a master to the SMBus host acting as a slave. It is the same form as Write Word, with the command code replaced by the alerting device's address.

```
[S] [HostAddr] [Wr] A [DevAddr] A [DataLow] A [DataHigh] A [P]
```

This is implemented in the following way in the Linux kernel:

- I2C bus drivers which support SMBus Host Notify should report I2C_FUNC_SMBUS_HOST_NOTIFY.
- I2C bus drivers trigger SMBus Host Notify by a call to i2c_handle_smbus_host_notify().
- I2C drivers for devices which can trigger SMBus Host Notify will have client->irq assigned to a Host Notify IRQ if noone else specified an other.

There is currently no way to retrieve the data parameter from the client.

1.3.14 Packet Error Checking (PEC)

Packet Error Checking was introduced in Revision 1.1 of the specification.

PEC adds a CRC-8 error-checking byte to transfers using it, immediately before the terminating STOP.

1.3.15 Address Resolution Protocol (ARP)

The Address Resolution Protocol was introduced in Revision 2.0 of the specification. It is a higher-layer protocol which uses the messages above.

ARP adds device enumeration and dynamic address assignment to the protocol. All ARP communications use slave address 0x61 and require PEC checksums.

1.3.16 SMBus Alert

SMBus Alert was introduced in Revision 1.0 of the specification.

The SMBus alert protocol allows several SMBus slave devices to share a single interrupt pin on the SMBus master, while still allowing the master to know which slave triggered the interrupt.

This is implemented the following way in the Linux kernel:

- I2C bus drivers which support SMBus alert should call i2c_new_smbus_alert_device() to install SMBus alert support.
- I2C drivers for devices which can trigger SMBus alerts should implement the optional alert() callback.

1.3.17 I2C Block Transactions

The following I2C block transactions are similar to the SMBus Block Read and Write operations, except these do not have a Count byte. They are supported by the SMBus layer and are described here for completeness, but they are NOT defined by the SMBus specification.

I2C block transactions do not limit the number of bytes transferred but the SMBus layer places a limit of 32 bytes.

1.3.18 I2C Block Read

Implemented by i2c_smbus_read_i2c_block_data()

This command reads a block of bytes from a device, from a designated register that is specified through the Comm byte:

```
S Addr Wr [A] Comm [A]
           S Addr Rd [A] [Data] A [Data] A ... A [Data] NA P
```

Functionality flag: I2C_FUNC_SMBUS_READ_I2C_BLOCK

1.3.19 I2C Block Write

Implemented by i2c_smbus_write_i2c_block_data()

The opposite of the Block Read command, this writes bytes to a device, to a designated register that is specified through the Comm byte. Note that command lengths of 0, 2, or more bytes are supported as they are indistinguishable from data.

```
S Addr Wr [A] Comm [A] Data [A] Data [A] ... [A] Data [A] P
```

Functionality flag: I2C_FUNC_SMBUS_WRITE_I2C_BLOCK

1.4 How to instantiate I2C devices

Unlike PCI or USB devices, I2C devices are not enumerated at the hardware level. Instead, the software must know which devices are connected on each I2C bus segment, and what address these devices are using. For this reason, the kernel code must instantiate I2C devices explicitly. There are several ways to achieve this, depending on the context and requirements.

1.4.1 Method 1: Declare the I2C devices statically

This method is appropriate when the I2C bus is a system bus as is the case for many embedded systems. On such systems, each I2C bus has a number which is known in advance. It is thus possible to pre-declare the I2C devices which live on this bus.

This information is provided to the kernel in a different way on different architectures: device tree, ACPI or board files.

When the I2C bus in question is registered, the I2C devices will be instantiated automatically by i2c-core. The devices will be automatically unbound and destroyed when the I2C bus they sit on goes away (if ever).

Declare the I2C devices via devicetree

On platforms using devicetree, the declaration of I2C devices is done in subnodes of the master controller.

Example:

```
i2c1: i2c@400a0000 {
        /* ... master properties skipped ... */
        clock-frequency = <100000>;

        flash@50 {
                compatible = "atmel,24c256";
                reg = <0x50>;
        };

        pca9532: gpio@60 {
                compatible = "nxp,pca9532";
                gpio-controller;
                #gpio-cells = <2>;
                reg = <0x60>;
        };
};
```

Here, two devices are attached to the bus using a speed of 100kHz. For additional properties which might be needed to set up the device, please refer to its devicetree documentation in Documentation/devicetree/bindings/.

Declare the I2C devices via ACPI

ACPI can also describe I2C devices. There is special documentation for this which is currently located at ../firmware-guide/acpi/enumeration.

Declare the I2C devices in board files

In many embedded architectures, devicetree has replaced the old hardware description based on board files, but the latter are still used in old code. Instantiating I2C devices via board files is done with an array of struct i2c_board_info which is registered by calling i2c_register_board_info().

Example (from omap2 h4):

```
static struct i2c_board_info h4_i2c_board_info[] __initdata = {
      {
              I2C_BOARD_INFO("isp1301_omap", 0x2d),
              .irq            = OMAP_GPIO_IRQ(125),
      },
      {       /* EEPROM on mainboard */
              I2C_BOARD_INFO("24c01", 0x52),
              .platform_data  = &m24c01,
      },
      {       /* EEPROM on cpu card */
              I2C_BOARD_INFO("24c01", 0x57),
              .platform_data  = &m24c01,
      },
};

static void __init omap_h4_init(void)
{
      (...)
      i2c_register_board_info(1, h4_i2c_board_info,
                      ARRAY_SIZE(h4_i2c_board_info));
      (...)
}
```

The above code declares 3 devices on I2C bus 1, including their respective addresses and custom data needed by their drivers.

1.4.2 Method 2: Instantiate the devices explicitly

This method is appropriate when a larger device uses an I2C bus for internal communication. A typical case is TV adapters. These can have a tuner, a video decoder, an audio decoder, etc. usually connected to the main chip by the means of an I2C bus. You won't know the number of the I2C bus in advance, so the method 1 described above can't be used. Instead, you can instantiate your I2C devices explicitly. This is done by filling a struct i2c_board_info and calling i2c_new_client_device().

Example (from the sfe4001 network driver):

```
static struct i2c_board_info sfe4001_hwmon_info = {
      I2C_BOARD_INFO("max6647", 0x4e),
};

int sfe4001_init(struct efx_nic *efx)
{
      (...)
      efx->board_info.hwmon_client =
```

(continues on next page)

(continued from previous page)

```
                i2c_new_client_device(&efx->i2c_adap, &sfe4001_hwmon_info);

        (...)
}
```

The above code instantiates 1 I2C device on the I2C bus which is on the network adapter in question.

A variant of this is when you don't know for sure if an I2C device is present or not (for example for an optional feature which is not present on cheap variants of a board but you have no way to tell them apart), or it may have different addresses from one board to the next (manufacturer changing its design without notice). In this case, you can call i2c_new_scanned_device() instead of i2c_new_client_device().

Example (from the nxp OHCI driver):

```
static const unsigned short normal_i2c[] = { 0x2c, 0x2d, I2C_CLIENT_END };

static int usb_hcd_nxp_probe(struct platform_device *pdev)
{
        (...)
        struct i2c_adapter *i2c_adap;
        struct i2c_board_info i2c_info;

        (...)
        i2c_adap = i2c_get_adapter(2);
        memset(&i2c_info, 0, sizeof(struct i2c_board_info));
        strscpy(i2c_info.type, "isp1301_nxp", sizeof(i2c_info.type));
        isp1301_i2c_client = i2c_new_scanned_device(i2c_adap, &i2c_info,
                                                    normal_i2c, NULL);
        i2c_put_adapter(i2c_adap);
        (...)
}
```

The above code instantiates up to 1 I2C device on the I2C bus which is on the OHCI adapter in question. It first tries at address 0x2c, if nothing is found there it tries address 0x2d, and if still nothing is found, it simply gives up.

The driver which instantiated the I2C device is responsible for destroying it on cleanup. This is done by calling i2c_unregister_device() on the pointer that was earlier returned by i2c_new_client_device() or i2c_new_scanned_device().

1.4.3 Method 3: Probe an I2C bus for certain devices

Sometimes you do not have enough information about an I2C device, not even to call i2c_new_scanned_device(). The typical case is hardware monitoring chips on PC mainboards. There are several dozen models, which can live at 25 different addresses. Given the huge number of mainboards out there, it is next to impossible to build an exhaustive list of the hardware monitoring chips being used. Fortunately, most of these chips have manufacturer and device ID registers, so they can be identified by probing.

In that case, I2C devices are neither declared nor instantiated explicitly. Instead,

i2c-core will probe for such devices as soon as their drivers are loaded, and if any is found, an I2C device will be instantiated automatically. In order to prevent any misbehavior of this mechanism, the following restrictions apply:

- The I2C device driver must implement the detect() method, which identifies a supported device by reading from arbitrary registers.
- Only buses which are likely to have a supported device and agree to be probed, will be probed. For example this avoids probing for hardware monitoring chips on a TV adapter.

Example: See lm90_driver and lm90_detect() in drivers/hwmon/lm90.c

I2C devices instantiated as a result of such a successful probe will be destroyed automatically when the driver which detected them is removed, or when the underlying I2C bus is itself destroyed, whichever happens first.

Those of you familiar with the I2C subsystem of 2.4 kernels and early 2.6 kernels will find out that this method 3 is essentially similar to what was done there. Two significant differences are:

- Probing is only one way to instantiate I2C devices now, while it was the only way back then. Where possible, methods 1 and 2 should be preferred. Method 3 should only be used when there is no other way, as it can have undesirable side effects.
- I2C buses must now explicitly say which I2C driver classes can probe them (by the means of the class bitfield), while all I2C buses were probed by default back then. The default is an empty class which means that no probing happens. The purpose of the class bitfield is to limit the aforementioned undesirable side effects.

Once again, method 3 should be avoided wherever possible. Explicit device instantiation (methods 1 and 2) is much preferred for it is safer and faster.

1.4.4 Method 4: Instantiate from user-space

In general, the kernel should know which I2C devices are connected and what addresses they live at. However, in certain cases, it does not, so a sysfs interface was added to let the user provide the information. This interface is made of 2 attribute files which are created in every I2C bus directory: `new_device` and `delete_device`. Both files are write only and you must write the right parameters to them in order to properly instantiate, respectively delete, an I2C device.

File `new_device` takes 2 parameters: the name of the I2C device (a string) and the address of the I2C device (a number, typically expressed in hexadecimal starting with 0x, but can also be expressed in decimal.)

File `delete_device` takes a single parameter: the address of the I2C device. As no two devices can live at the same address on a given I2C segment, the address is sufficient to uniquely identify the device to be deleted.

Example:

```
# echo eeprom 0x50 > /sys/bus/i2c/devices/i2c-3/new_device
```

While this interface should only be used when in-kernel device declaration can't be done, there is a variety of cases where it can be helpful:

- The I2C driver usually detects devices (method 3 above) but the bus segment your device lives on doesn't have the proper class bit set and thus detection doesn't trigger.
- The I2C driver usually detects devices, but your device lives at an unexpected address.
- The I2C driver usually detects devices, but your device is not detected, either because the detection routine is too strict, or because your device is not officially supported yet but you know it is compatible.
- You are developing a driver on a test board, where you soldered the I2C device yourself.

This interface is a replacement for the force_* module parameters some I2C drivers implement. Being implemented in i2c-core rather than in each device driver individually, it is much more efficient, and also has the advantage that you do not have to reload the driver to change a setting. You can also instantiate the device before the driver is loaded or even available, and you don't need to know what driver the device needs.

1.5 I2C Bus Drivers

1.5.1 Kernel driver i2c-ali1535

Supported adapters:

- Acer Labs, Inc. ALI 1535 (south bridge)

  Datasheet: Now under NDA http://www.ali.com.tw/

Authors:

- Frodo Looijaard <frodol@dds.nl>,
- Philip Edelbrock <phil@netroedge.com>,
- Mark D. Studebaker <mdsxyz123@yahoo.com>,
- Dan Eaton <dan.eaton@rocketlogix.com>,
- Stephen Rousset<stephen.rousset@rocketlogix.com>

Description

This is the driver for the SMB Host controller on Acer Labs Inc. (ALI) M1535 South Bridge.

The M1535 is a South bridge for portable systems. It is very similar to the M15x3 South bridges also produced by Acer Labs Inc. Some of the registers within the part have moved and some have been redefined slightly. Additionally, the sequencing of the SMBus transactions has been modified to be more consistent with the sequencing recommended by the manufacturer and observed through

testing. These changes are reflected in this driver and can be identified by comparing this driver to the i2c-ali15x3 driver. For an overview of these chips see http://www.acerlabs.com

The SMB controller is part of the M7101 device, which is an ACPI-compliant Power Management Unit (PMU).

The whole M7101 device has to be enabled for the SMB to work. You can't just enable the SMB alone. The SMB and the ACPI have separate I/O spaces. We make sure that the SMB is enabled. We leave the ACPI alone.

Features

This driver controls the SMB Host only. This driver does not use interrupts.

1.5.2 Kernel driver i2c-ali1563

Supported adapters:

- Acer Labs, Inc. ALI 1563 (south bridge)

  Datasheet: Now under NDA http://www.ali.com.tw/

Author: Patrick Mochel <mochel@digitalimplant.org>

Description

This is the driver for the SMB Host controller on Acer Labs Inc. (ALI) M1563 South Bridge.

For an overview of these chips see http://www.acerlabs.com

The M1563 southbridge is deceptively similar to the M1533, with a few notable exceptions. One of those happens to be the fact they upgraded the i2c core to be SMBus 2.0 compliant, and happens to be almost identical to the i2c controller found in the Intel 801 south bridges.

Features

This driver controls the SMB Host only. This driver does not use interrupts.

1.5.3 Kernel driver i2c-ali15x3

Supported adapters:

- Acer Labs, Inc. ALI 1533 and 1543C (south bridge)

  Datasheet: Now under NDA http://www.ali.com.tw/

Authors:

- Frodo Looijaard <frodol@dds.nl>,
- Philip Edelbrock <phil@netroedge.com>,

- Mark D. Studebaker <mdsxyz123@yahoo.com>

Module Parameters

- **force_addr: int** Initialize the base address of the i2c controller

Notes

The force_addr parameter is useful for boards that don't set the address in the BIOS. Does not do a PCI force; the device must still be present in lspci. Don't use this unless the driver complains that the base address is not set.

Example:

```
modprobe i2c-ali15x3 force_addr=0xe800
```

SMBus periodically hangs on ASUS P5A motherboards and can only be cleared by a power cycle. Cause unknown (see Issues below).

Description

This is the driver for the SMB Host controller on Acer Labs Inc. (ALI) M1541 and M1543C South Bridges.

The M1543C is a South bridge for desktop systems.

The M1541 is a South bridge for portable systems.

They are part of the following ALI chipsets:

- "Aladdin Pro 2" includes the M1621 Slot 1 North bridge with AGP and 100MHz CPU Front Side bus
- "Aladdin V" includes the M1541 Socket 7 North bridge with AGP and 100MHz CPU Front Side bus

  Some Aladdin V motherboards:

  - Asus P5A
  - Atrend ATC-5220
  - BCM/GVC VP1541
  - Biostar M5ALA
  - Gigabyte GA-5AX (Generally doesn't work because the BIOS doesn't enable the 7101 device!)
  - Iwill XA100 Plus
  - Micronics C200
  - Microstar (MSI) MS-5169
- "Aladdin IV" includes the M1541 Socket 7 North bridge with host bus up to 83.3 MHz.

For an overview of these chips see http://www.acerlabs.com. At this time the full data sheets on the web site are password protected, however if you contact the ALI office in San Jose they may give you the password.

The M1533/M1543C devices appear as FOUR separate devices on the PCI bus. An output of lspci will show something similar to the following:

```
00:02.0 USB Controller: Acer Laboratories Inc. M5237 (rev 03)
00:03.0 Bridge: Acer Laboratories Inc. M7101      <= THIS IS THE ONE WE␣
↪NEED
00:07.0 ISA bridge: Acer Laboratories Inc. M1533 (rev c3)
00:0f.0 IDE interface: Acer Laboratories Inc. M5229 (rev c1)
```

Important: If you have a M1533 or M1543C on the board and you get "ali15x3: Error: Can't detect ali15x3!" then run lspci.

If you see the 1533 and 5229 devices but NOT the 7101 device, then you must enable ACPI, the PMU, SMB, or something similar in the BIOS.

The driver won't work if it can't find the M7101 device.

The SMB controller is part of the M7101 device, which is an ACPI-compliant Power Management Unit (PMU).

The whole M7101 device has to be enabled for the SMB to work. You can't just enable the SMB alone. The SMB and the ACPI have separate I/O spaces. We make sure that the SMB is enabled. We leave the ACPI alone.

Features

This driver controls the SMB Host only. The SMB Slave controller on the M15X3 is not enabled. This driver does not use interrupts.

Issues

This driver requests the I/O space for only the SMB registers. It doesn't use the ACPI region.

On the ASUS P5A motherboard, there are several reports that the SMBus will hang and this can only be resolved by powering off the computer. It appears to be worse when the board gets hot, for example under heavy CPU load, or in the summer. There may be electrical problems on this board. On the P5A, the W83781D sensor chip is on both the ISA and SMBus. Therefore the SMBus hangs can generally be avoided by accessing the W83781D on the ISA bus only.

1.5.4 Kernel driver i2c-amd756

Supported adapters:

- AMD 756
- AMD 766
- AMD 768
- AMD 8111

  Datasheets: Publicly available on AMD website

- nVidia nForce

  Datasheet: Unavailable

Authors:

- Frodo Looijaard <frodol@dds.nl>,
- Philip Edelbrock <phil@netroedge.com>

Description

This driver supports the AMD 756, 766, 768 and 8111 Peripheral Bus Controllers, and the nVidia nForce.

Note that for the 8111, there are two SMBus adapters. The SMBus 1.0 adapter is supported by this driver, and the SMBus 2.0 adapter is supported by the i2c-amd8111 driver.

1.5.5 Kernel driver i2c-adm8111

Supported adapters:

- AMD-8111 SMBus 2.0 PCI interface

Datasheets: AMD datasheet not yet available, but almost everything can be found in the publicly available ACPI 2.0 specification, which the adapter follows.

Author: Vojtech Pavlik <vojtech@suse.cz>

Description

If you see something like this:

```
00:07.2 SMBus: Advanced Micro Devices [AMD] AMD-8111 SMBus 2.0 (rev 02)
        Subsystem: Advanced Micro Devices [AMD] AMD-8111 SMBus 2.0
        Flags: medium devsel, IRQ 19
        I/O ports at d400 [size=32]
```

in your `lspci -v`, then this driver is for your chipset.

Process Call Support

Supported.

SMBus 2.0 Support

Supported. Both PEC and block process call support is implemented. Slave mode or host notification are not yet implemented.

Notes

Note that for the 8111, there are two SMBus adapters. The SMBus 2.0 adapter is supported by this driver, and the SMBus 1.0 adapter is supported by the i2c-amd756 driver.

1.5.6 Kernel driver i2c-amd-mp2

Supported adapters:

- AMD MP2 PCIe interface

Datasheet: not publicly available.

Authors:

- Shyam Sundar S K <Shyam-sundar.S-k@amd.com>
- Nehal Shah <nehal-bakulchandra.shah@amd.com>
- Elie Morisse <syniurge@gmail.com>

Description

The MP2 is an ARM processor programmed as an I2C controller and communicating with the x86 host through PCI.

If you see something like this:

```
03:00.7 MP2 I2C controller: Advanced Micro Devices, Inc. [AMD] Device 15e6
```

in your `lspci -v`, then this driver is for your device.

1.5.7 Kernel driver i2c-diolan-u2c

Supported adapters:

- Diolan U2C-12 I2C-USB adapter

  Documentation: http://www.diolan.com/i2c/u2c12.html

Author: Guenter Roeck <linux@roeck-us.net>

Description

This is the driver for the Diolan U2C-12 USB-I2C adapter.

The Diolan U2C-12 I2C-USB Adapter provides a low cost solution to connect a computer to I2C slave devices using a USB interface. It also supports connectivity to SPI devices.

This driver only supports the I2C interface of U2C-12. The driver does not use interrupts.

Module parameters

- frequency: I2C bus frequency

1.5.8 Kernel driver i2c-i801

Supported adapters:

- Intel 82801AA and 82801AB (ICH and ICH0 - part of the '810' and '810E' chipsets)
- Intel 82801BA (ICH2 - part of the '815E' chipset)
- Intel 82801CA/CAM (ICH3)
- Intel 82801DB (ICH4) (HW PEC supported)
- Intel 82801EB/ER (ICH5) (HW PEC supported)
- Intel 6300ESB
- Intel 82801FB/FR/FW/FRW (ICH6)
- Intel 82801G (ICH7)
- Intel 631xESB/632xESB (ESB2)
- Intel 82801H (ICH8)
- Intel 82801I (ICH9)
- Intel EP80579 (Tolapai)
- Intel 82801JI (ICH10)
- Intel 5/3400 Series (PCH)
- Intel 6 Series (PCH)
- Intel Patsburg (PCH)
- Intel DH89xxCC (PCH)
- Intel Panther Point (PCH)
- Intel Lynx Point (PCH)
- Intel Avoton (SOC)
- Intel Wellsburg (PCH)

- Intel Coleto Creek (PCH)
- Intel Wildcat Point (PCH)
- Intel BayTrail (SOC)
- Intel Braswell (SOC)
- Intel Sunrise Point (PCH)
- Intel Kaby Lake (PCH)
- Intel DNV (SOC)
- Intel Broxton (SOC)
- Intel Lewisburg (PCH)
- Intel Gemini Lake (SOC)
- Intel Cannon Lake (PCH)
- Intel Cedar Fork (PCH)
- Intel Ice Lake (PCH)
- Intel Comet Lake (PCH)
- Intel Elkhart Lake (PCH)
- Intel Tiger Lake (PCH)
- Intel Jasper Lake (SOC)

  Datasheets: Publicly available at the Intel website

On Intel Patsburg and later chipsets, both the normal host SMBus controller and the additional 'Integrated Device Function' controllers are supported.

Authors:

- Mark Studebaker <mdsxyz123@yahoo.com>
- Jean Delvare <jdelvare@suse.de>

Module Parameters

- disable_features (bit vector)

Disable selected features normally supported by the device. This makes it possible to work around possible driver or hardware bugs if the feature in question doesn't work as intended for whatever reason. Bit values:

0x01	disable SMBus PEC
0x02	disable the block buffer
0x08	disable the I2C block read functionality
0x10	don't use interrupts
0x20	disable SMBus Host Notify

Description

The ICH (properly known as the 82801AA), ICH0 (82801AB), ICH2 (82801BA), ICH3 (82801CA/CAM) and later devices (PCH) are Intel chips that are a part of Intel's '810' chipset for Celeron-based PCs, '810E' chipset for Pentium-based PCs, '815E' chipset, and others.

The ICH chips contain at least SEVEN separate PCI functions in TWO logical PCI devices. An output of lspci will show something similar to the following:

```
00:1e.0 PCI bridge: Intel Corporation: Unknown device 2418 (rev 01)
00:1f.0 ISA bridge: Intel Corporation: Unknown device 2410 (rev 01)
00:1f.1 IDE interface: Intel Corporation: Unknown device 2411 (rev 01)
00:1f.2 USB Controller: Intel Corporation: Unknown device 2412 (rev 01)
00:1f.3 Unknown class [0c05]: Intel Corporation: Unknown device 2413 (rev␣
↪01)
```

The SMBus controller is function 3 in device 1f. Class 0c05 is SMBus Serial Controller.

The ICH chips are quite similar to Intel's PIIX4 chip, at least in the SMBus controller.

Process Call Support

Block process call is supported on the 82801EB (ICH5) and later chips.

I2C Block Read Support

I2C block read is supported on the 82801EB (ICH5) and later chips.

SMBus 2.0 Support

The 82801DB (ICH4) and later chips support several SMBus 2.0 features.

Interrupt Support

PCI interrupt support is supported on the 82801EB (ICH5) and later chips.

Hidden ICH SMBus

If your system has an Intel ICH south bridge, but you do NOT see the SMBus device at 00:1f.3 in lspci, and you can't figure out any way in the BIOS to enable it, it means it has been hidden by the BIOS code. Asus is well known for first doing this on their P4B motherboard, and many other boards after that. Some vendor machines are affected as well.

The first thing to try is the "i2c-scmi" ACPI driver. It could be that the SMBus was hidden on purpose because it'll be driven by ACPI. If the i2c-scmi driver works for you, just forget about the i2c-i801 driver and don't try to unhide the ICH SMBus.

Even if i2c-scmi doesn't work, you better make sure that the SMBus isn't used by the ACPI code. Try loading the "fan" and "thermal" drivers, and check in /sys/class/thermal. If you find a thermal zone with type "acpitz", it's likely that the ACPI is accessing the SMBus and it's safer not to unhide it. Only once you are certain that ACPI isn't using the SMBus, you can attempt to unhide it.

In order to unhide the SMBus, we need to change the value of a PCI register before the kernel enumerates the PCI devices. This is done in drivers/pci/quirks.c, where all affected boards must be listed (see function asus_hides_smbus_hostbridge.) If the SMBus device is missing, and you think there's something interesting on the SMBus (e.g. a hardware monitoring chip), you need to add your board to the list.

The motherboard is identified using the subvendor and subdevice IDs of the host bridge PCI device. Get yours with `lspci -n -v -s 00:00.0`:

```
00:00.0 Class 0600: 8086:2570 (rev 02)
        Subsystem: 1043:80f2
        Flags: bus master, fast devsel, latency 0
        Memory at fc000000 (32-bit, prefetchable) [size=32M]
        Capabilities: [e4] #09 [2106]
        Capabilities: [a0] AGP version 3.0
```

Here the host bridge ID is 2570 (82865G/PE/P), the subvendor ID is 1043 (Asus) and the subdevice ID is 80f2 (P4P800-X). You can find the symbolic names for the bridge ID and the subvendor ID in include/linux/pci_ids.h, and then add a case for your subdevice ID at the right place in drivers/pci/quirks.c. Then please give it very good testing, to make sure that the unhidden SMBus doesn't conflict with e.g. ACPI.

If it works, proves useful (i.e. there are usable chips on the SMBus) and seems safe, please submit a patch for inclusion into the kernel.

Note: There's a useful script in lm_sensors 2.10.2 and later, named unhide_ICH_SMBus (in prog/hotplug), which uses the fakephp driver to temporarily unhide the SMBus without having to patch and recompile your kernel. It's very convenient if you just want to check if there's anything interesting on your hidden ICH SMBus.

The lm_sensors project gratefully acknowledges the support of Texas Instruments in the initial development of this driver.

The lm_sensors project gratefully acknowledges the support of Intel in the development of SMBus 2.0 / ICH4 features of this driver.

1.5.9 Kernel driver i2c-ismt

Supported adapters:

- Intel S12xx series SOCs

Authors: Bill Brown <bill.e.brown@intel.com>

Module Parameters

- bus_speed (unsigned int)

Allows changing of the bus speed. Normally, the bus speed is set by the BIOS and never needs to be changed. However, some SMBus analyzers are too slow for monitoring the bus during debug, thus the need for this module parameter. Specify the bus speed in kHz.

Available bus frequency settings:

0	no change
80	kHz
100	kHz
400	kHz
1000	kHz

Description

The S12xx series of SOCs have a pair of integrated SMBus 2.0 controllers targeted primarily at the microserver and storage markets.

The S12xx series contain a pair of PCI functions. An output of lspci will show something similar to the following:

```
00:13.0 System peripheral: Intel Corporation Centerton SMBus 2.0␣
↪Controller 0
00:13.1 System peripheral: Intel Corporation Centerton SMBus 2.0␣
↪Controller 1
```

1.5.10 Driver i2c-mlxcpld

Author: Michael Shych <michaelsh@mellanox.com>

This is the Mellanox I2C controller logic, implemented in Lattice CPLD device.

Device supports:

- Master mode.
- One physical bus.
- Polling mode.

This controller is equipped within the next Mellanox systems: “msx6710” , “msx6720” , “msb7700” , “msn2700” , “msx1410” , “msn2410” , “msb7800” , “msn2740” , “msn2100” .

The next transaction types are supported:

- Receive Byte/Block.
- Send Byte/Block.
- Read Byte/Block.

- Write Byte/Block.

Registers:

CPBLTY	0x0	• **capability reg.** Bits [6:5] - transaction length. b01 - 72B is supported, 36B in other case. Bit 7 - SMBus block read support.
CTRL	0x1	• **control reg.** Resets all the registers.
HALF_CYC	0x4	• **cycle reg.** Configure the width of I2C SCL half clock cycle (in 4 LPC_CLK units).
I2C_HOLD	0x5	• **hold reg.** OE (output enable) is delayed by value set to this register (in LPC_CLK units)
CMD		0x6 - command reg. Bit 0, 0 = write, 1 = read. Bits [7:1] - the 7bit Address of the I2C device. It should be written last as it triggers an I2C transaction.
NUM_DATA	0x7	• **data size reg.** Number of data bytes to write in read transaction
NUM_ADDR	0x8	• **address reg.** Number of address bytes to write in read transaction
STATUS	0x9	• **status reg.** Bit 0 -

1.5.11 Kernel driver i2c-nforce2

Supported adapters:

- nForce2 MCP 10de:0064
- nForce2 Ultra 400 MCP 10de:0084
- nForce3 Pro150 MCP 10de:00D4
- nForce3 250Gb MCP 10de:00E4
- nForce4 MCP 10de:0052
- nForce4 MCP-04 10de:0034
- nForce MCP51 10de:0264
- nForce MCP55 10de:0368
- nForce MCP61 10de:03EB
- nForce MCP65 10de:0446
- nForce MCP67 10de:0542
- nForce MCP73 10de:07D8
- nForce MCP78S 10de:0752
- nForce MCP79 10de:0AA2

Datasheet: not publicly available, but seems to be similar to the AMD-8111 SMBus 2.0 adapter.

Authors:

- Hans-Frieder Vogt <hfvogt@gmx.net>,
- Thomas Leibold <thomas@plx.com>,
- Patrick Dreker <patrick@dreker.de>

Description

i2c-nforce2 is a driver for the SMBuses included in the nVidia nForce2 MCP.

If your `lspci -v` listing shows something like the following:

```
00:01.1 SMBus: nVidia Corporation: Unknown device 0064 (rev a2)
        Subsystem: Asustek Computer, Inc.: Unknown device 0c11
        Flags: 66Mhz, fast devsel, IRQ 5
        I/O ports at c000 [size=32]
        Capabilities: <available only to root>
```

then this driver should support the SMBuses of your motherboard.

Notes

The SMBus adapter in the nForce2 chipset seems to be very similar to the SMBus 2.0 adapter in the AMD-8111 south bridge. However, I could only get the driver to work with direct I/O access, which is different to the EC interface of the AMD-8111. Tested on Asus A7N8X. The ACPI DSDT table of the Asus A7N8X lists two SMBuses, both of which are supported by this driver.

1.5.12 Kernel driver i2c-nvidia-gpu

Datasheet: not publicly available.

Authors: Ajay Gupta <ajayg@nvidia.com>

Description

i2c-nvidia-gpu is a driver for I2C controller included in NVIDIA Turing and later GPUs and it is used to communicate with Type-C controller on GPUs.

If your `lspci -v` listing shows something like the following:

```
01:00.3 Serial bus controller [0c80]: NVIDIA Corporation Device 1ad9 (rev␣
↪a1)
```

then this driver should support the I2C controller of your GPU.

1.5.13 Kernel driver i2c-ocores

Supported adapters:

- OpenCores.org I2C controller by Richard Herveille (see datasheet link) https://opencores.org/project/i2c/overview

Author: Peter Korsgaard <peter@korsgaard.com>

Description

i2c-ocores is an i2c bus driver for the OpenCores.org I2C controller IP core by Richard Herveille.

Usage

i2c-ocores uses the platform bus, so you need to provide a struct platform_device with the base address and interrupt number. The dev.platform_data of the device should also point to a struct ocores_i2c_platform_data (see linux/platform_data/i2c-ocores.h) describing the distance between registers and the input clock speed. There is also a possibility to attach a list of i2c_board_info which the i2c-ocores driver will add to the bus upon creation.

E.G. something like:

```
static struct resource ocores_resources[] = {
      [0] = {
              .start  = MYI2C_BASEADDR,
              .end    = MYI2C_BASEADDR + 8,
              .flags  = IORESOURCE_MEM,
      },
      [1] = {
              .start  = MYI2C_IRQ,
              .end    = MYI2C_IRQ,
              .flags  = IORESOURCE_IRQ,
      },
};

/* optional board info */
struct i2c_board_info ocores_i2c_board_info[] = {
      {
              I2C_BOARD_INFO("tsc2003", 0x48),
              .platform_data = &tsc2003_platform_data,
              .irq = TSC_IRQ
      },
      {
              I2C_BOARD_INFO("adv7180", 0x42 >> 1),
              .irq = ADV_IRQ
      }
};

static struct ocores_i2c_platform_data myi2c_data = {
      .regstep        = 2,            /* two bytes between registers */
      .clock_khz      = 50000,        /* input clock of 50MHz */
      .devices        = ocores_i2c_board_info, /* optional table of␣
↪devices */
      .num_devices    = ARRAY_SIZE(ocores_i2c_board_info), /* table size */
};

static struct platform_device myi2c = {
      .name                   = "ocores-i2c",
      .dev = {
              .platform_data  = &myi2c_data,
      },
      .num_resources          = ARRAY_SIZE(ocores_resources),
      .resource               = ocores_resources,
};
```

1.5.14 Kernel driver i2c-parport

Author: Jean Delvare <jdelvare@suse.de>

This is a unified driver for several i2c-over-parallel-port adapters, such as the ones made by Philips, Velleman or ELV. This driver is meant as a replacement for the older, individual drivers:

- i2c-philips-par
- i2c-elv
- i2c-velleman

- video/i2c-parport (NOT the same as this one, dedicated to home brew teletext adapters)

It currently supports the following devices:

- (type=0) Philips adapter
- (type=1) home brew teletext adapter
- (type=2) Velleman K8000 adapter
- (type=3) ELV adapter
- (type=4) Analog Devices ADM1032 evaluation board
- (type=5) Analog Devices evaluation boards: ADM1025, ADM1030, ADM1031
- (type=6) Barco LPT->DVI (K5800236) adapter
- (type=7) One For All JP1 parallel port adapter
- (type=8) VCT-jig

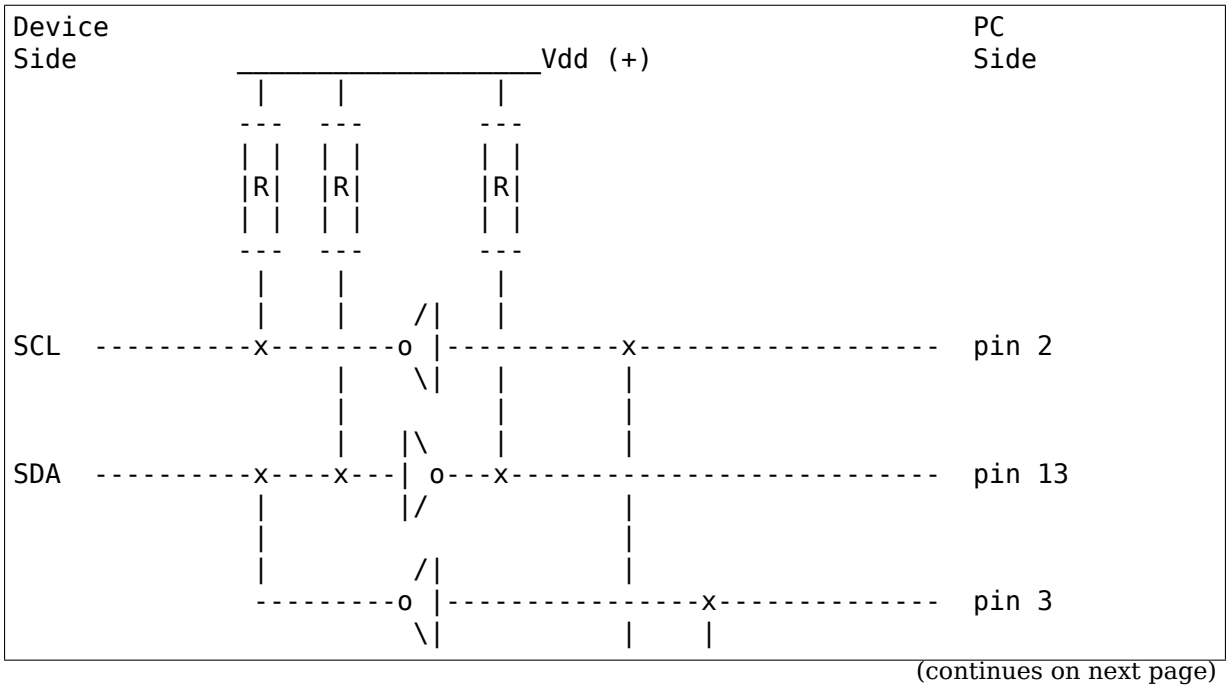
These devices use different pinout configurations, so you have to tell the driver what you have, using the type module parameter. There is no way to autodetect the devices. Support for different pinout configurations can be easily added when needed.

Earlier kernels defaulted to type=0 (Philips). But now, if the type parameter is missing, the driver will simply fail to initialize.

SMBus alert support is available on adapters which have this line properly connected to the parallel port's interrupt pin.

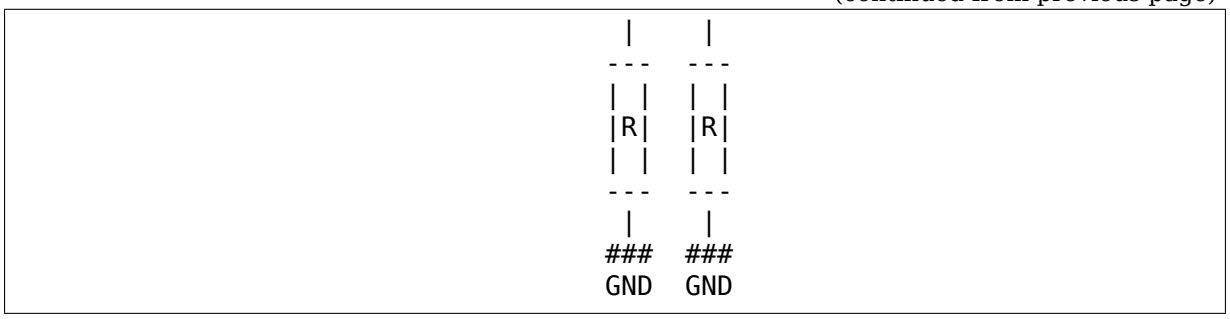
Building your own adapter

If you want to build you own i2c-over-parallel-port adapter, here is a sample electronics schema (credits go to Sylvain Munaut):



(continues on next page)

(continued from previous page)

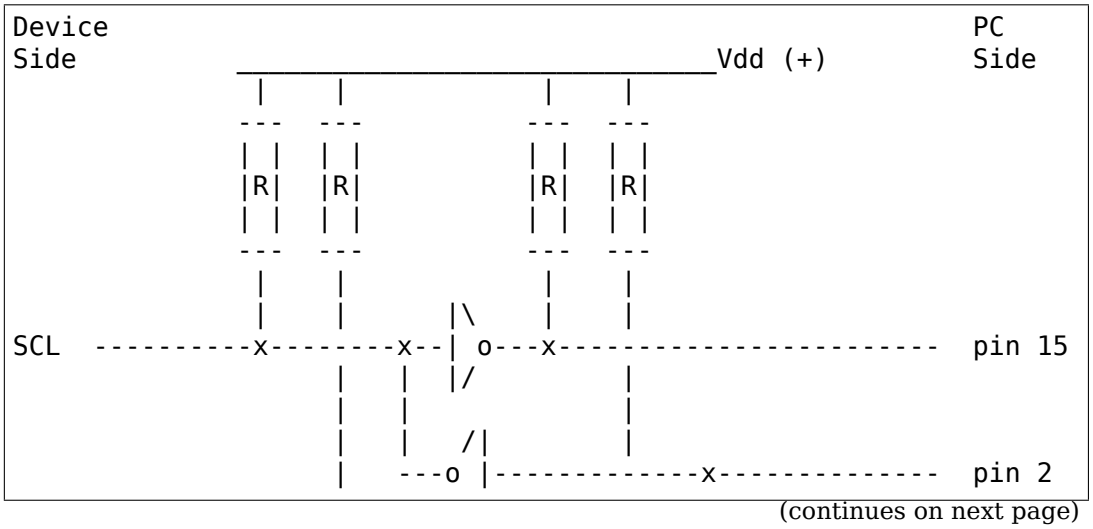


Remarks:

- This is the exact pinout and electronics used on the Analog Devices evaluation boards.
- All inverters:

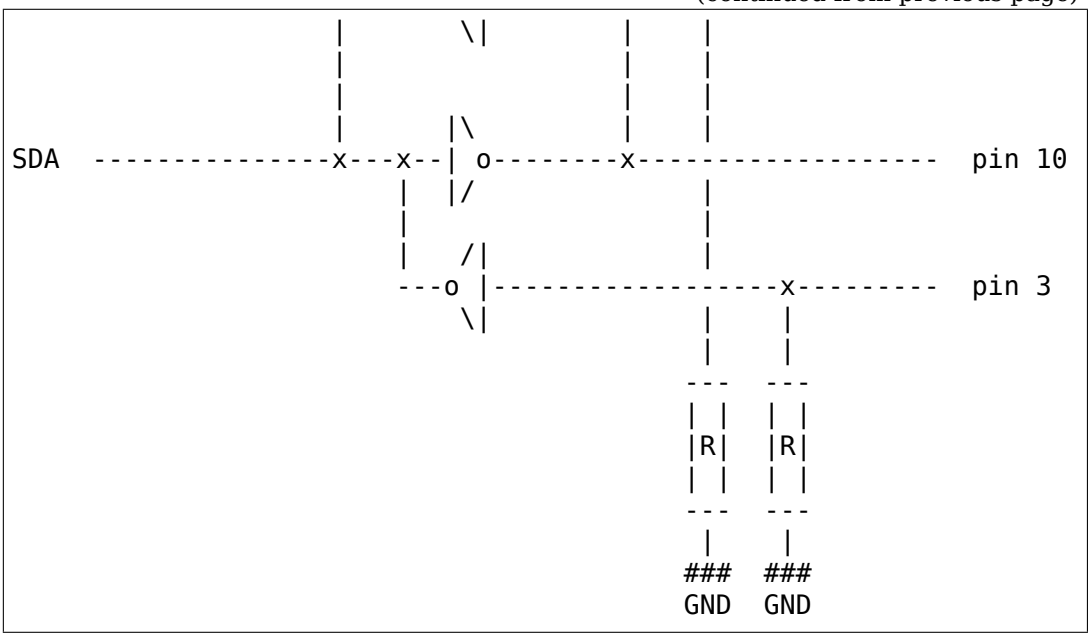
  must be 74HC05, they must be open collector output.

- All resitors are 10k.
- Pins 18-25 of the parallel port connected to GND.
- Pins 4-9 (D2-D7) could be used as VDD is the driver drives them high. The ADM1032 evaluation board uses D4-D7. Beware that the amount of current you can draw from the parallel port is limited. Also note that all connected lines MUST BE driven at the same state, else you'll short circuit the output buffers! So plugging the I2C adapter after loading the i2c-parport module might be a good safety since data line state prior to init may be unknown.
- This is 5V!
- Obviously you cannot read SCL (so it's not really standard-compliant). Pretty easy to add, just copy the SDA part and use another input pin. That would give (ELV compatible pinout):



(continues on next page)

(continued from previous page)



If possible, you should use the same pinout configuration as existing adapters do, so you won't even have to change the code.

Similar (but different) drivers

This driver is NOT the same as the i2c-pport driver found in the i2c package. The i2c-pport driver makes use of modern parallel port features so that you don't need additional electronics. It has other restrictions however, and was not ported to Linux 2.6 (yet).

This driver is also NOT the same as the i2c-pcf-epp driver found in the lm_sensors package. The i2c-pcf-epp driver doesn't use the parallel port as an I2C bus directly. Instead, it uses it to control an external I2C bus master. That driver was not ported to Linux 2.6 (yet) either.

Legacy documentation for Velleman adapter

Useful links:

- Velleman http://www.velleman.be/
- Velleman K8000 Howto http://howto.htlw16.ac.at/k8000-howto.html

The project has lead to new libs for the Velleman K8000 and K8005:

LIBK8000 v1.99.1 and LIBK8005 v0.21

With these libs, you can control the K8000 interface card and the K8005 stepper motor card with the simple commands which are in the original Velleman software, like SetIOchannel, ReadADchannel, SendStepCCWFull and many more, using /dev/velleman.

- http://home.wanadoo.nl/hihihi/libk8000.htm
- http://home.wanadoo.nl/hihihi/libk8005.htm

- http://struyve.mine.nu:8080/index.php?block=k8000
- http://sourceforge.net/projects/libk8005/

One For All JP1 parallel port adapter

The JP1 project revolves around a set of remote controls which expose the I2C bus their internal configuration EEPROM lives on via a 6 pin jumper in the battery compartment. More details can be found at:

http://www.hifi-remote.com/jp1/

Details of the simple parallel port hardware can be found at:

http://www.hifi-remote.com/jp1/hardware.shtml

1.5.15 Kernel driver i2c-pca-isa

Supported adapters:

This driver supports ISA boards using the Philips PCA 9564 Parallel bus to I2C bus controller

Author: Ian Campbell <icampbell@arcom.com>, Arcom Control Systems

Module Parameters

- **base int** I/O base address
- **irq int** IRQ interrupt
- **clock int** Clock rate as described in table 1 of PCA9564 datasheet

Description

This driver supports ISA boards using the Philips PCA 9564 Parallel bus to I2C bus controller

1.5.16 Kernel driver i2c-piix4

Supported adapters:

- Intel 82371AB PIIX4 and PIIX4E
- Intel 82443MX (440MX) Datasheet: Publicly available at the Intel website
- ServerWorks OSB4, CSB5, CSB6, HT-1000 and HT-1100 southbridges Datasheet: Only available via NDA from ServerWorks
- ATI IXP200, IXP300, IXP400, SB600, SB700 and SB800 southbridges Datasheet: Not publicly available SB700 register reference available at: http://support.amd.com/us/Embedded_TechDocs/43009_sb7xx_rrg_pub_1.00.pdf

- AMD SP5100 (SB700 derivative found on some server mainboards) Datasheet: Publicly available at the AMD website http://support.amd.com/us/Embedded_TechDocs/44413.pdf
- AMD Hudson-2, ML, CZ Datasheet: Not publicly available
- Hygon CZ Datasheet: Not publicly available
- Standard Microsystems (SMSC) SLC90E66 (Victory66) southbridge Datasheet: Publicly available at the SMSC website http://www.smsc.com

Authors:

- Frodo Looijaard <frodol@dds.nl>
- Philip Edelbrock <phil@netroedge.com>

Module Parameters

- force: int Forcibly enable the PIIX4. DANGEROUS!
- force_addr: int Forcibly enable the PIIX4 at the given address. EXTREMELY DANGEROUS!

Description

The PIIX4 (properly known as the 82371AB) is an Intel chip with a lot of functionality. Among other things, it implements the PCI bus. One of its minor functions is implementing a System Management Bus. This is a true SMBus - you can not access it on I2C levels. The good news is that it natively understands SMBus commands and you do not have to worry about timing problems. The bad news is that non-SMBus devices connected to it can confuse it mightily. Yes, this is known to happen…

Do `lspci -v` and see whether it contains an entry like this:

```
0000:00:02.3 Bridge: Intel Corp. 82371AB/EB/MB PIIX4 ACPI (rev 02)
             Flags: medium devsel, IRQ 9
```

Bus and device numbers may differ, but the function number must be identical (like many PCI devices, the PIIX4 incorporates a number of different 'functions', which can be considered as separate devices). If you find such an entry, you have a PIIX4 SMBus controller.

On some computers (most notably, some Dells), the SMBus is disabled by default. If you use the insmod parameter 'force=1', the kernel module will try to enable it. THIS IS VERY DANGEROUS! If the BIOS did not set up a correct address for this module, you could get in big trouble (read: crashes, data corruption, etc.). Try this only as a last resort (try BIOS updates first, for example), and backup first! An even more dangerous option is 'force_addr=<IOPORT>'. This will not only enable the PIIX4 like 'force' foes, but it will also set a new base I/O port address. The SMBus parts of the PIIX4 needs a range of 8 of these addresses to function correctly. If these addresses are already reserved by some other device, you will get into big trouble! DON'T USE THIS IF YOU ARE NOT VERY SURE ABOUT WHAT YOU ARE DOING!

The PIIX4E is just an new version of the PIIX4; it is supported as well. The PIIX/PIIX3 does not implement an SMBus or I2C bus, so you can't use this driver on those mainboards.

The ServerWorks Southbridges, the Intel 440MX, and the Victory66 are identical to the PIIX4 in I2C/SMBus support.

The AMD SB700, SB800, SP5100 and Hudson-2 chipsets implement two PIIX4-compatible SMBus controllers. If your BIOS initializes the secondary controller, it will be detected by this driver as an "Auxiliary SMBus Host Controller".

If you own Force CPCI735 motherboard or other OSB4 based systems you may need to change the SMBus Interrupt Select register so the SMBus controller uses the SMI mode.

1) Use lspci command and locate the PCI device with the SMBus controller: 00:0f.0 ISA bridge: ServerWorks OSB4 South Bridge (rev 4f) The line may vary for different chipsets. Please consult the driver source for all possible PCI ids (and lspci -n to match them). Lets assume the device is located at 00:0f.0.

2) Now you just need to change the value in 0xD2 register. Get it first with command: lspci -xxx -s 00:0f.0 If the value is 0x3 then you need to change it to 0x1: setpci -s 00:0f.0 d2.b=1

Please note that you don't need to do that in all cases, just when the SMBus is not working properly.

Hardware-specific issues

This driver will refuse to load on IBM systems with an Intel PIIX4 SMBus. Some of these machines have an RFID EEPROM (24RF08) connected to the SMBus, which can easily get corrupted due to a state machine bug. These are mostly Thinkpad laptops, but desktop systems may also be affected. We have no list of all affected systems, so the only safe solution was to prevent access to the SMBus on all IBM systems (detected using DMI data.)

For additional information, read: http://www.lm-sensors.org/browser/lm-sensors/trunk/README

1.5.17 Kernel driver i2c-sis5595

Authors:

- Frodo Looijaard <frodol@dds.nl>,
- Mark D. Studebaker <mdsxyz123@yahoo.com>,
- Philip Edelbrock <phil@netroedge.com>

Supported adapters:

- Silicon Integrated Systems Corp. SiS5595 Southbridge Datasheet: Publicly available at the Silicon Integrated Systems Corp. site.

Note: all have mfr. ID 0x1039.

SUPPORTED	PCI ID
5595	0008

Note: these chips contain a 0008 device which is incompatible with the 5595. We recognize these by the presence of the listed "blacklist" PCI ID and refuse to load.

NOT SUPPORTED	PCI ID	BLACKLIST PCI ID
540	0008	0540
550	0008	0550
5513	0008	5511
5581	0008	5597
5582	0008	5597
5597	0008	5597
5598	0008	5597/5598
630	0008	0630
645	0008	0645
646	0008	0646
648	0008	0648
650	0008	0650
651	0008	0651
730	0008	0730
735	0008	0735
745	0008	0745
746	0008	0746

Module Parameters

force_addr=0x1234	Set the I/O base address. Useful for boards that don't set the address in the BIOS. Does not do a PCI force; the device must still be present in lspci. Don't use this unless the driver complains that the base address is not set.

Description

i2c-sis5595 is a true SMBus host driver for motherboards with the SiS5595 south-bridges.

WARNING: If you are trying to access the integrated sensors on the SiS5595 chip, you want the sis5595 driver for those, not this driver. This driver is a BUS driver, not a CHIP driver. A BUS driver is used by other CHIP drivers to access chips on the bus.

1.5.18 Kernel driver i2c-sis630

Supported adapters:

- **Silicon Integrated Systems Corp (SiS)** 630 chipset (Datasheet: available at http://www.sfr-fresh.com/linux) 730 chipset 964 chipset
- Possible other SiS chipsets ?

Author:

- Alexander Malysh <amalysh@web.de>
- Amaury Decrême <amaury.decreme@gmail.com> - SiS964 support

Module Parameters

force = [1|0]	Forcibly enable the SIS630. DANGEROUS! This can be interesting for chipsets not named above to check if it works for you chipset, but DANGEROUS!
high_clock = [1|0]	Forcibly set Host Master Clock to 56KHz (default, what your BIOS use). DANGEROUS! This should be a bit faster, but freeze some systems (i.e. my Laptop). SIS630/730 chip only.

Description

This SMBus only driver is known to work on motherboards with the above named chipsets.

If you see something like this:

```
00:00.0 Host bridge: Silicon Integrated Systems [SiS] 630 Host (rev 31)
00:01.0 ISA bridge: Silicon Integrated Systems [SiS] 85C503/5513
```

or like this:

```
00:00.0 Host bridge: Silicon Integrated Systems [SiS] 730 Host (rev 02)
00:01.0 ISA bridge: Silicon Integrated Systems [SiS] 85C503/5513
```

or like this:

```
00:00.0 Host bridge: Silicon Integrated Systems [SiS] 760/M760 Host (rev␣
↪02)
00:02.0 ISA bridge: Silicon Integrated Systems [SiS] SiS964 [MuTIOL Media␣
↪IO]
                                                    LPC Controller (rev␣
↪36)
```

in your `lspci` output , then this driver is for your chipset.

Thank You

Philip Edelbrock <phil@netroedge.com> - testing SiS730 support Mark M. Hoffman <mhoffman@lightlink.com> - bug fixes

To anyone else which I forgot here ;), thanks!

1.5.19 Kernel driver i2c-sis96x

Replaces 2.4.x i2c-sis645

Supported adapters:

- Silicon Integrated Systems Corp (SiS)

  Any combination of these host bridges: 645, 645DX (aka 646), 648, 650, 651, 655, 735, 745, 746

  and these south bridges: 961, 962, 963(L)

Author: Mark M. Hoffman <mhoffman@lightlink.com>

Description

This SMBus only driver is known to work on motherboards with the above named chipset combinations. The driver was developed without benefit of a proper datasheet from SiS. The SMBus registers are assumed compatible with those of the SiS630, although they are located in a completely different place. Thanks to Alexander Malysh <amalysh@web.de> for providing the SiS630 datasheet (and driver).

The command `lspci` as root should produce something like these lines:

```
00:00.0 Host bridge: Silicon Integrated Systems [SiS]: Unknown device 0645
00:02.0 ISA bridge: Silicon Integrated Systems [SiS] 85C503/5513
00:02.1 SMBus: Silicon Integrated Systems [SiS]: Unknown device 0016
```

or perhaps this:

```
00:00.0 Host bridge: Silicon Integrated Systems [SiS]: Unknown device 0645
00:02.0 ISA bridge: Silicon Integrated Systems [SiS]: Unknown device 0961
00:02.1 SMBus: Silicon Integrated Systems [SiS]: Unknown device 0016
```

(kernel versions later than 2.4.18 may fill in the “Unknown” s)

If you can’ t see it please look on quirk_sis_96x_smbus (drivers/pci/quirks.c) (also if southbridge detection fails)

I suspect that this driver could be made to work for the following SiS chipsets as well: 635, and 635T. If anyone owns a board with those chips AND is willing to risk crashing & burning an otherwise well-behaved kernel in the name of progress··· please contact me at <mhoffman@lightlink.com> or via the linux-i2c mailing list: <linux-i2c@vger.kernel.org>. Please send bug reports and/or success stories as well.

TO DOs

- The driver does not support SMBus block reads/writes; I may add them if a scenario is found where they' re needed.

Thank You

Mark D. Studebaker <mdsxyz123@yahoo.com>

- design hints and bug fixes

Alexander Maylsh <amalysh@web.de>

- ditto, plus an important datasheet···almost the one I really wanted

Hans-Günter Lütke Uphues <hg_lu@t-online.de>

- patch for SiS735

Robert Zwerus <arzie@dds.nl>

- testing for SiS645DX

Kianusch Sayah Karadji <kianusch@sk-tech.net>

- patch for SiS645DX/962

Ken Healy

- patch for SiS655

To anyone else who has written w/ feedback, thanks!

1.5.20 Kernel driver i2c-taos-evm

Author: Jean Delvare <jdelvare@suse.de>

This is a driver for the evaluation modules for TAOS I2C/SMBus chips. The modules include an SMBus master with limited capabilities, which can be controlled over the serial port. Virtually all evaluation modules are supported, but a few lines of code need to be added for each new module to instantiate the right I2C chip on the bus. Obviously, a driver for the chip in question is also needed.

Currently supported devices are:

- TAOS TSL2550 EVM

For additional information on TAOS products, please see http://www.taosinc.com/

Using this driver

In order to use this driver, you' ll need the serport driver, and the inputattach tool, which is part of the input-utils package. The following commands will tell the kernel that you have a TAOS EVM on the first serial port:

```
# modprobe serport
# inputattach --taos-evm /dev/ttyS0
```

Technical details

Only 4 SMBus transaction types are supported by the TAOS evaluation modules: * Receive Byte * Send Byte * Read Byte * Write Byte

The communication protocol is text-based and pretty simple. It is described in a PDF document on the CD which comes with the evaluation module. The communication is rather slow, because the serial port has to operate at 1200 bps. However, I don' t think this is a big concern in practice, as these modules are meant for evaluation and testing only.

1.5.21 Kernel driver i2c-viapro

Supported adapters:

- VIA Technologies, Inc. VT82C596A/B Datasheet: Sometimes available at the VIA website
- VIA Technologies, Inc. VT82C686A/B Datasheet: Sometimes available at the VIA website
- VIA Technologies, Inc. VT8231, VT8233, VT8233A Datasheet: available on request from VIA
- VIA Technologies, Inc. VT8235, VT8237R, VT8237A, VT8237S, VT8251 Datasheet: available on request and under NDA from VIA
- VIA Technologies, Inc. CX700 Datasheet: available on request and under NDA from VIA
- VIA Technologies, Inc. VX800/VX820 Datasheet: available on http://linux.via.com.tw
- VIA Technologies, Inc. VX855/VX875 Datasheet: available on http://linux.via.com.tw
- VIA Technologies, Inc. VX900 Datasheet: available on http://linux.via.com.tw

Authors:

- Kyösti Mälkki <kmalkki@cc.hut.fi>,
- Mark D. Studebaker <mdsxyz123@yahoo.com>,
- Jean Delvare <jdelvare@suse.de>

Module Parameters

- force: int Forcibly enable the SMBus controller. DANGEROUS!
- force_addr: int Forcibly enable the SMBus at the given address. EXTREMELY DANGEROUS!

Description

i2c-viapro is a true SMBus host driver for motherboards with one of the supported VIA south bridges.

Your `lspci -n` listing must show one of these :

device 1106:3050	(VT82C596A function 3)
device 1106:3051	(VT82C596B function 3)
device 1106:3057	(VT82C686 function 4)
device 1106:3074	(VT8233)
device 1106:3147	(VT8233A)
device 1106:8235	(VT8231 function 4)
device 1106:3177	(VT8235)
device 1106:3227	(VT8237R)
device 1106:3337	(VT8237A)
device 1106:3372	(VT8237S)
device 1106:3287	(VT8251)
device 1106:8324	(CX700)
device 1106:8353	(VX800/VX820)
device 1106:8409	(VX855/VX875)
device 1106:8410	(VX900)

If none of these show up, you should look in the BIOS for settings like enable ACPI / SMBus or even USB.

Except for the oldest chips (VT82C596A/B, VT82C686A and most probably VT8231), this driver supports I2C block transactions. Such transactions are mainly useful to read from and write to EEPROMs.

The CX700/VX800/VX820 additionally appears to support SMBus PEC, although this driver doesn't implement it yet.

1.5.22 Kernel driver i2c-via

Supported adapters:

- VIA Technologies, InC. VT82C586B Datasheet: Publicly available at the VIA website

Author: Kyösti Mälkki <kmalkki@cc.hut.fi>

Description

i2c-via is an i2c bus driver for motherboards with VIA chipset.

The following VIA pci chipsets are supported:

- MVP3, VP3, VP2/97, VPX/97
- others with South bridge VT82C586B

Your `lspci` listing must show this

```
Bridge: VIA Technologies, Inc. VT82C586B ACPI (rev 10)
```

Problems?

Q: You have VT82C586B on the motherboard, but not in the listing.

A: Go to your BIOS setup, section PCI devices or similar. Turn USB support on, and try again.

Q: No error messages, but still i2c doesn't seem to work.

A: This can happen. This driver uses the pins VIA recommends in their datasheets, but there are several ways the motherboard manufacturer can actually wire the lines.

1.5.23 Kernel driver scx200_acb

Author: Christer Weinigel <wingel@nano-system.com>

The driver supersedes the older, never merged driver named i2c-nscacb.

Module Parameters

- base: up to 4 ints Base addresses for the ACCESS.bus controllers on SCx200 and SC1100 devices

  By default the driver uses two base addresses 0x820 and 0x840. If you want only one base address, specify the second as 0 so as to override this default.

Description

Enable the use of the ACCESS.bus controller on the Geode SCx200 and SC1100 processors and the CS5535 and CS5536 Geode companion devices.

Device-specific notes

The SC1100 WRAP boards are known to use base addresses 0x810 and 0x820. If the scx200_acb driver is built into the kernel, add the following parameter to your boot command line:

```
scx200_acb.base=0x810,0x820
```

If the scx200_acb driver is built as a module, add the following line to a configuration file in /etc/modprobe.d/ instead:

```
options scx200_acb base=0x810,0x820
```

1.6 I2C muxes and complex topologies

There are a couple of reasons for building more complex I2C topologies than a straight-forward I2C bus with one adapter and one or more devices.

1. A mux may be needed on the bus to prevent address collisions.
2. The bus may be accessible from some external bus master, and arbitration may be needed to determine if it is ok to access the bus.
3. A device (particularly RF tuners) may want to avoid the digital noise from the I2C bus, at least most of the time, and sits behind a gate that has to be operated before the device can be accessed.

1.6.1 Etc

These constructs are represented as I2C adapter trees by Linux, where each adapter has a parent adapter (except the root adapter) and zero or more child adapters. The root adapter is the actual adapter that issues I2C transfers, and all adapters with a parent are part of an "i2c-mux" object (quoted, since it can also be an arbitrator or a gate).

Depending of the particular mux driver, something happens when there is an I2C transfer on one of its child adapters. The mux driver can obviously operate a mux, but it can also do arbitration with an external bus master or open a gate. The mux driver has two operations for this, select and deselect. select is called before the transfer and (the optional) deselect is called after the transfer.

1.6.2 Locking

There are two variants of locking available to I2C muxes, they can be mux-locked or parent-locked muxes. As is evident from below, it can be useful to know if a mux is mux-locked or if it is parent-locked. The following list was correct at the time of writing:

In drivers/i2c/muxes/:

i2c-arb-gpio-challenge	Parent-locked
i2c-mux-gpio	Normally parent-locked, mux-locked iff all involved gpio pins are controlled by the same I2C root adapter that they mux.
i2c-mux-gpmux	Normally parent-locked, mux-locked iff specified in device-tree.
i2c-mux-ltc4306	Mux-locked
i2c-mux-mlxcpld	Parent-locked
i2c-mux-pca9541	Parent-locked
i2c-mux-pca954x	Parent-locked
i2c-mux-pinctrl	Normally parent-locked, mux-locked iff all involved pinctrl devices are controlled by the same I2C root adapter that they mux.
i2c-mux-reg	Parent-locked

In drivers/iio/:

gyro/mpu3050	Mux-locked
imu/inv_mpu6050/	Mux-locked

In drivers/media/:

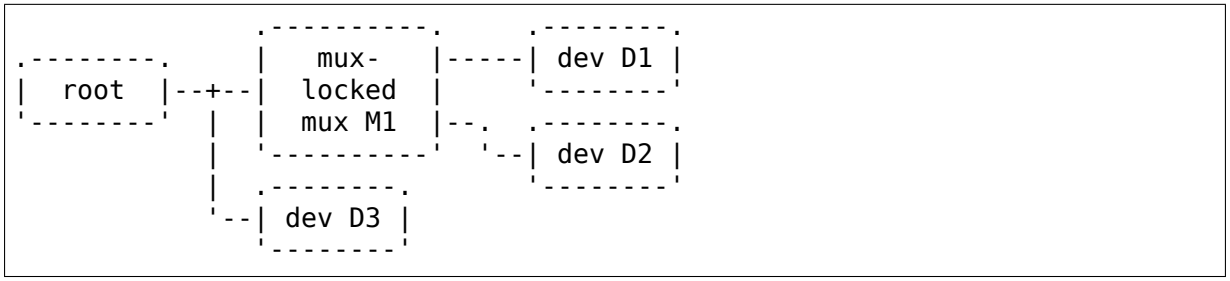
dvb-frontends/lgdt3306a	Mux-locked
dvb-frontends/m88ds3103	Parent-locked
dvb-frontends/rtl2830	Parent-locked
dvb-frontends/rtl2832	Mux-locked
dvb-frontends/si2168	Mux-locked
usb/cx231xx/	Parent-locked

Mux-locked muxes

Mux-locked muxes does not lock the entire parent adapter during the full select-transfer-deselect transaction, only the muxes on the parent adapter are locked. Mux-locked muxes are mostly interesting if the select and/or deselect operations must use I2C transfers to complete their tasks. Since the parent adapter is not fully locked during the full transaction, unrelated I2C transfers may interleave the different stages of the transaction. This has the benefit that the mux driver may be easier and cleaner to implement, but it has some caveats.

ML1.	If you build a topology with a mux-locked mux being the parent of a parent-locked mux, this might break the expectation from the parent-locked mux that the root adapter is locked during the transaction.
ML2.	It is not safe to build arbitrary topologies with two (or more) mux-locked muxes that are not siblings, when there are address collisions between the devices on the child adapters of these non-sibling muxes. I.e. the select-transfer-deselect transaction targeting e.g. device address 0x42 behind mux-one may be interleaved with a similar operation targeting device address 0x42 behind mux-two. The intension with such a topology would in this hypothetical example be that mux-one and mux-two should not be selected simultaneously, but mux-locked muxes do not guarantee that in all topologies.
ML3.	A mux-locked mux cannot be used by a driver for auto-closing gates/muxes, i.e. something that closes automatically after a given number (one, in most cases) of I2C transfers. Unrelated I2C transfers may creep in and close prematurely.
ML4.	If any non-I2C operation in the mux driver changes the I2C mux state, the driver has to lock the root adapter during that operation. Otherwise garbage may appear on the bus as seen from devices behind the mux, when an unrelated I2C transfer is in flight during the non-I2C mux-changing operation.

Mux-locked Example



When there is an access to D1, this happens:

1. Someone issues an I2C transfer to D1.
2. M1 locks muxes on its parent (the root adapter in this case).
3. M1 calls ->select to ready the mux.
4. M1 (presumably) does some I2C transfers as part of its select. These transfers are normal I2C transfers that locks the parent adapter.
5. M1 feeds the I2C transfer from step 1 to its parent adapter as a normal I2C transfer that locks the parent adapter.
6. M1 calls ->deselect, if it has one.
7. Same rules as in step 4, but for ->deselect.
8. M1 unlocks muxes on its parent.

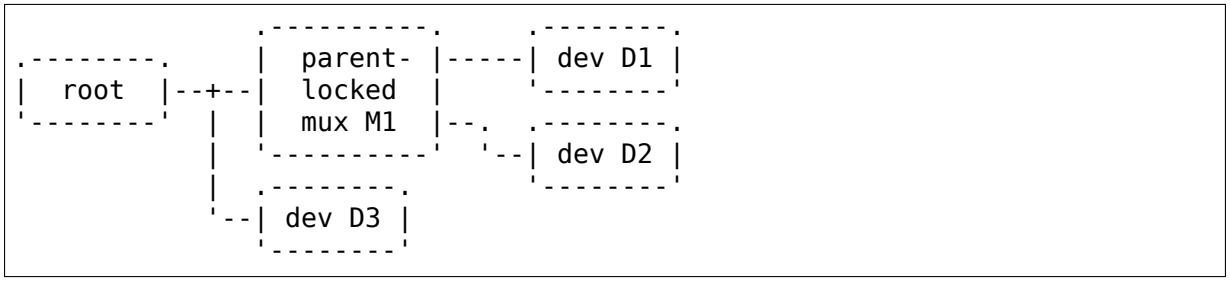
This means that accesses to D2 are lockout out for the full duration of the entire operation. But accesses to D3 are possibly interleaved at any point.

Parent-locked muxes

Parent-locked muxes lock the parent adapter during the full select- transfer-deselect transaction. The implication is that the mux driver has to ensure that any and all I2C transfers through that parent adapter during the transaction are unlocked I2C transfers (using e.g. __i2c_transfer), or a deadlock will follow. There are a couple of caveats.

PL1.	If you build a topology with a parent-locked mux being the child of another mux, this might break a possible assumption from the child mux that the root adapter is unused between its select op and the actual transfer (e.g. if the child mux is auto-closing and the parent mux issues I2C transfers as part of its select). This is especially the case if the parent mux is mux-locked, but it may also happen if the parent mux is parent-locked.
PL2.	If select/deselect calls out to other subsystems such as gpio, pinctrl, regmap or iio, it is essential that any I2C transfers caused by these subsystems are unlocked. This can be convoluted to accomplish, maybe even impossible if an acceptably clean solution is sought.

Parent-locked Example



When there is an access to D1, this happens:

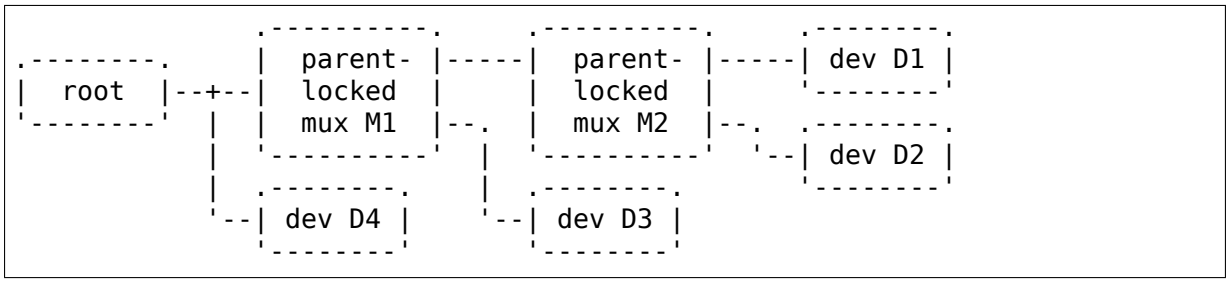
1. Someone issues an I2C transfer to D1.
2. M1 locks muxes on its parent (the root adapter in this case).
3. M1 locks its parent adapter.
4. M1 calls ->select to ready the mux.
5. If M1 does any I2C transfers (on this root adapter) as part of its select, those transfers must be unlocked I2C transfers so that they do not deadlock the root adapter.
6. M1 feeds the I2C transfer from step 1 to the root adapter as an unlocked I2C transfer, so that it does not deadlock the parent adapter.
7. M1 calls ->deselect, if it has one.
8. Same rules as in step 5, but for ->deselect.
9. M1 unlocks its parent adapter.
10. M1 unlocks muxes on its parent.

This means that accesses to both D2 and D3 are locked out for the full duration of the entire operation.

1.6.3 Complex Examples

Parent-locked mux as parent of parent-locked mux

This is a useful topology, but it can be bad:

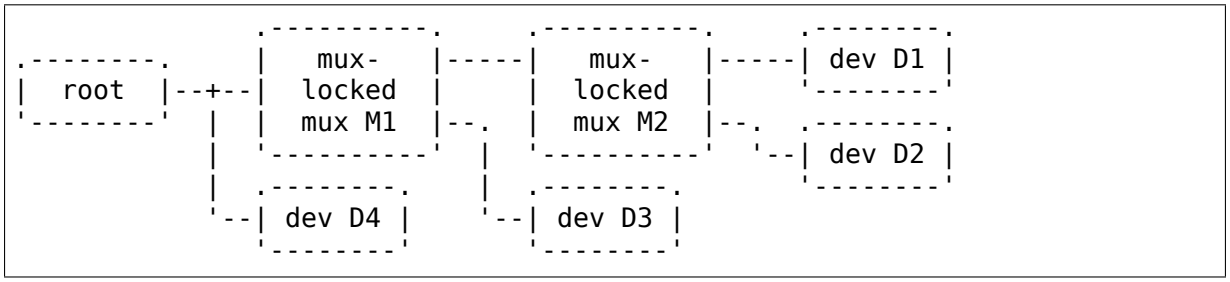


When any device is accessed, all other devices are locked out for the full duration of the operation (both muxes lock their parent, and specifically when M2 requests its parent to lock, M1 passes the buck to the root adapter).

This topology is bad if M2 is an auto-closing mux and M1->select issues any unlocked I2C transfers on the root adapter that may leak through and be seen by the M2 adapter, thus closing M2 prematurely.

Mux-locked mux as parent of mux-locked mux

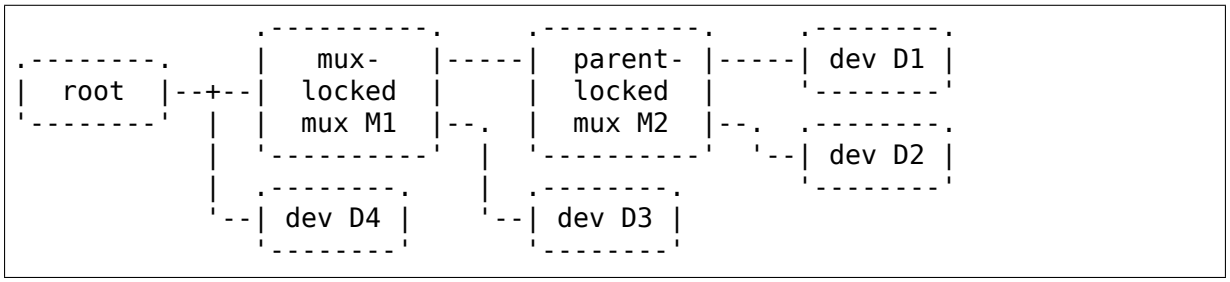
This is a good topology:



When device D1 is accessed, accesses to D2 are locked out for the full duration of the operation (muxes on the top child adapter of M1 are locked). But accesses to D3 and D4 are possibly interleaved at any point. Accesses to D3 locks out D1 and D2, but accesses to D4 are still possibly interleaved.

Mux-locked mux as parent of parent-locked mux

This is probably a bad topology:



When device D1 is accessed, accesses to D2 and D3 are locked out for the full duration of the operation (M1 locks child muxes on the root adapter). But accesses to D4 are possibly interleaved at any point.

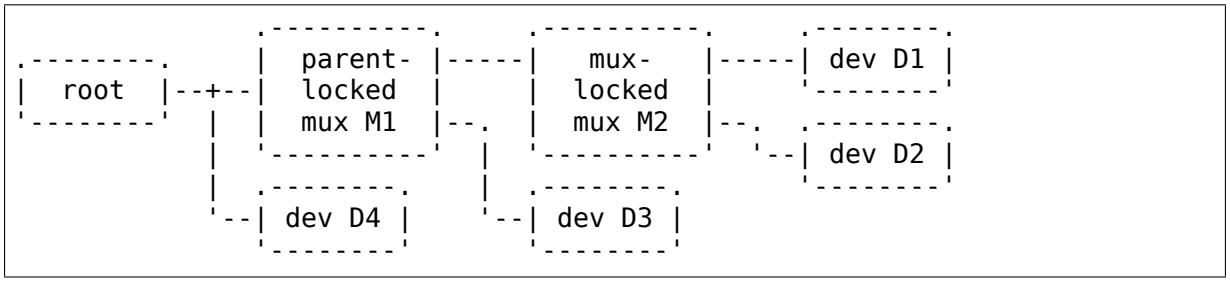
This kind of topology is generally not suitable and should probably be avoided. The reason is that M2 probably assumes that there will be no I2C transfers during its calls to ->select and ->deselect, and if there are, any such transfers might appear on the slave side of M2 as partial I2C transfers, i.e. garbage or worse. This might cause device lockups and/or other problems.

The topology is especially troublesome if M2 is an auto-closing mux. In that case, any interleaved accesses to D4 might close M2 prematurely, as might any I2C transfers part of M1->select.

But if M2 is not making the above stated assumption, and if M2 is not auto-closing, the topology is fine.

Parent-locked mux as parent of mux-locked mux

This is a good topology:

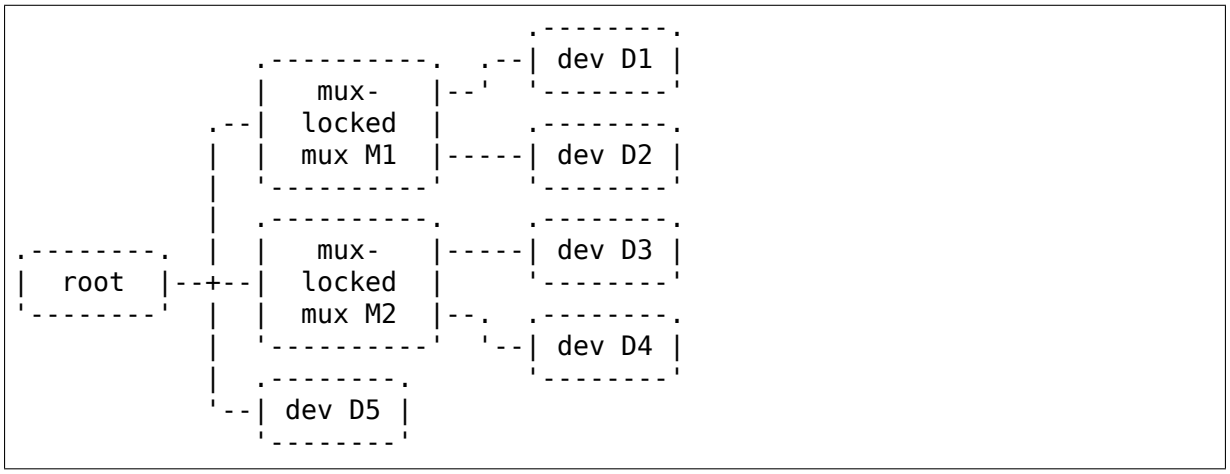


When D1 is accessed, accesses to D2 are locked out for the full duration of the operation (muxes on the top child adapter of M1 are locked). Accesses to D3 and D4 are possibly interleaved at any point, just as is expected for mux-locked muxes.

When D3 or D4 are accessed, everything else is locked out. For D3 accesses, M1 locks the root adapter. For D4 accesses, the root adapter is locked directly.

Two mux-locked sibling muxes

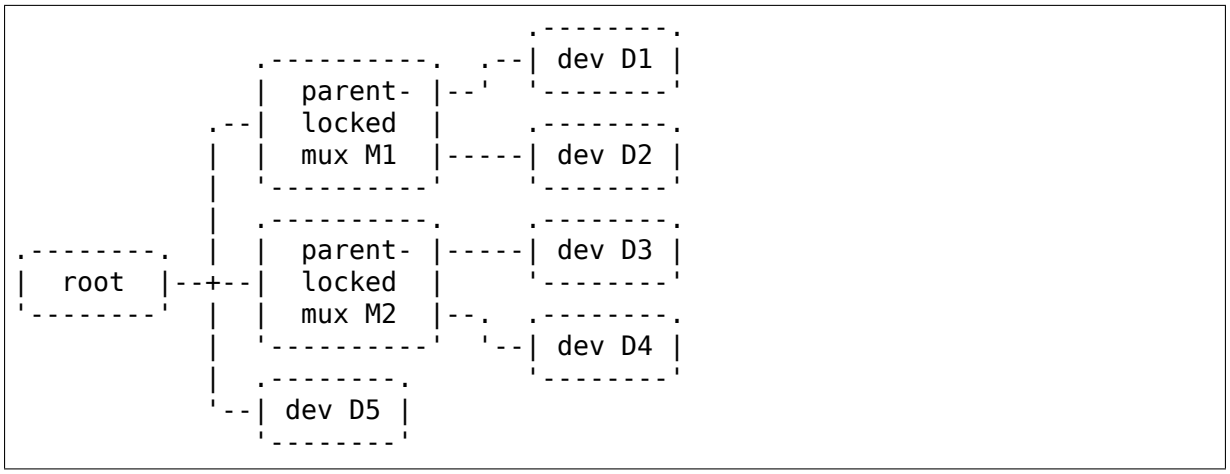
This is a good topology:



When D1 is accessed, accesses to D2, D3 and D4 are locked out. But accesses to D5 may be interleaved at any time.

Two parent-locked sibling muxes

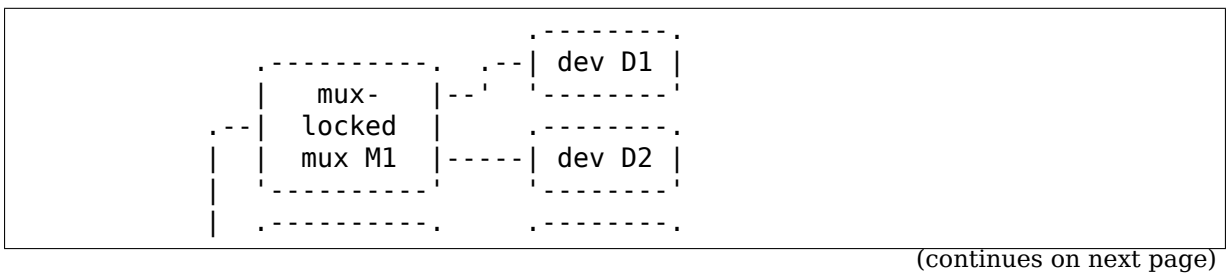
This is a good topology:



When any device is accessed, accesses to all other devices are locked out.

Mux-locked and parent-locked sibling muxes

This is a good topology:



(continues on next page)

(continued from previous page)

```
.--------.  |  |  parent- |-----| dev D3 |
|  root  |--+--|  locked  |     '--------'
'--------'  |  |  mux M2  |--.  .--------.
            |  '----------'  '--| dev D4 |
            |  .--------.       '--------'
            '--| dev D5 |
               '--------'
```

When D1 or D2 are accessed, accesses to D3 and D4 are locked out while accesses to D5 may interleave. When D3 or D4 are accessed, accesses to all other devices are locked out.

1.7 Kernel driver i2c-mux-gpio

Author: Peter Korsgaard <peter.korsgaard@barco.com>

1.7.1 Description

i2c-mux-gpio is an i2c mux driver providing access to I2C bus segments from a master I2C bus and a hardware MUX controlled through GPIO pins.

E.G.:

```
 ----------                ----------  Bus segment 1   - - - - -
|          | SCL/SDA    |          |--------------- |           |
|          |------------|          |
|          |            |          | Bus segment 2 |           |
|  Linux   | GPIO 1..N  |   MUX    |----------------   Devices
|          |------------|          |                |           |
|          |            |          | Bus segment M
|          |            |          |----------------|           |
 ----------                ----------                  - - - - -
```

SCL/SDA of the master I2C bus is multiplexed to bus segment 1..M according to the settings of the GPIO pins 1..N.

1.7.2 Usage

i2c-mux-gpio uses the platform bus, so you need to provide a struct platform_device with the platform_data pointing to a struct i2c_mux_gpio_platform_data with the I2C adapter number of the master bus, the number of bus segments to create and the GPIO pins used to control it. See include/linux/platform_data/i2c-mux-gpio.h for details.

E.G. something like this for a MUX providing 4 bus segments controlled through 3 GPIO pins:

```
#include <linux/platform_data/i2c-mux-gpio.h>
#include <linux/platform_device.h>

```

(continues on next page)

(continued from previous page)

```
static const unsigned myboard_gpiomux_gpios[] = {
      AT91_PIN_PC26, AT91_PIN_PC25, AT91_PIN_PC24
};

static const unsigned myboard_gpiomux_values[] = {
      0, 1, 2, 3
};

static struct i2c_mux_gpio_platform_data myboard_i2cmux_data = {
      .parent         = 1,
      .base_nr        = 2, /* optional */
      .values         = myboard_gpiomux_values,
      .n_values       = ARRAY_SIZE(myboard_gpiomux_values),
      .gpios          = myboard_gpiomux_gpios,
      .n_gpios        = ARRAY_SIZE(myboard_gpiomux_gpios),
      .idle           = 4, /* optional */
};

static struct platform_device myboard_i2cmux = {
      .name           = "i2c-mux-gpio",
      .id             = 0,
      .dev            = {
              .platform_data  = &myboard_i2cmux_data,
      },
};
```

If you don’t know the absolute GPIO pin numbers at registration time, you can instead provide a chip name (.chip_name) and relative GPIO pin numbers, and the i2c-mux-gpio driver will do the work for you, including deferred probing if the GPIO chip isn’t immediately available.

1.7.3 Device Registration

When registering your i2c-mux-gpio device, you should pass the number of any GPIO pin it uses as the device ID. This guarantees that every instance has a different ID.

Alternatively, if you don’t need a stable device name, you can simply pass PLATFORM_DEVID_AUTO as the device ID, and the platform core will assign a dynamic ID to your device. If you do not know the absolute GPIO pin numbers at registration time, this is even the only option.

CHAPTER

TWO

WRITING DEVICE DRIVERS

2.1 Implementing I2C device drivers

This is a small guide for those who want to write kernel drivers for I2C or SMBus devices, using Linux as the protocol host/master (not slave).

To set up a driver, you need to do several things. Some are optional, and some things can be done slightly or completely different. Use this as a guide, not as a rule book!

2.1.1 General remarks

Try to keep the kernel namespace as clean as possible. The best way to do this is to use a unique prefix for all global symbols. This is especially important for exported symbols, but it is a good idea to do it for non-exported symbols too. We will use the prefix `foo_` in this tutorial.

2.1.2 The driver structure

Usually, you will implement a single driver structure, and instantiate all clients from it. Remember, a driver structure contains general access routines, and should be zero-initialized except for fields with data you provide. A client structure holds device-specific information like the driver model device node, and its I2C address.

```
static struct i2c_device_id foo_idtable[] = {
      { "foo", my_id_for_foo },
      { "bar", my_id_for_bar },
      { }
};

MODULE_DEVICE_TABLE(i2c, foo_idtable);

static struct i2c_driver foo_driver = {
      .driver = {
              .name   = "foo",
              .pm     = &foo_pm_ops,  /* optional */
      },

      .id_table       = foo_idtable,
      .probe          = foo_probe,
```

(continues on next page)

(continued from previous page)

```
      .remove         = foo_remove,
      /* if device autodetection is needed: */
      .class          = I2C_CLASS_SOMETHING,
      .detect         = foo_detect,
      .address_list   = normal_i2c,

      .shutdown       = foo_shutdown, /* optional */
      .command        = foo_command,  /* optional, deprecated */
}
```

The name field is the driver name, and must not contain spaces. It should match the module name (if the driver can be compiled as a module), although you can use MODULE_ALIAS (passing "foo" in this example) to add another name for the module. If the driver name doesn't match the module name, the module won't be automatically loaded (hotplug/coldplug).

All other fields are for call-back functions which will be explained below.

2.1.3 Extra client data

Each client structure has a special `data` field that can point to any structure at all. You should use this to keep device-specific data.

```
/* store the value */
void i2c_set_clientdata(struct i2c_client *client, void *data);

/* retrieve the value */
void *i2c_get_clientdata(const struct i2c_client *client);
```

Note that starting with kernel 2.6.34, you don't have to set the `data` field to NULL in remove() or if probe() failed anymore. The i2c-core does this automatically on these occasions. Those are also the only times the core will touch this field.

2.1.4 Accessing the client

Let's say we have a valid client structure. At some time, we will need to gather information from the client, or write new information to the client.

I have found it useful to define foo_read and foo_write functions for this. For some cases, it will be easier to call the I2C functions directly, but many chips have some kind of register-value idea that can easily be encapsulated.

The below functions are simple examples, and should not be copied literally:

```
int foo_read_value(struct i2c_client *client, u8 reg)
{
      if (reg < 0x10) /* byte-sized register */
              return i2c_smbus_read_byte_data(client, reg);
      else            /* word-sized register */
              return i2c_smbus_read_word_data(client, reg);
}
```

(continues on next page)

(continued from previous page)

```
int foo_write_value(struct i2c_client *client, u8 reg, u16 value)
{
      if (reg == 0x10)        /* Impossible to write - driver error! */
              return -EINVAL;
      else if (reg < 0x10)    /* byte-sized register */
              return i2c_smbus_write_byte_data(client, reg, value);
      else                    /* word-sized register */
              return i2c_smbus_write_word_data(client, reg, value);
}
```

2.1.5 Probing and attaching

The Linux I2C stack was originally written to support access to hardware monitoring chips on PC motherboards, and thus used to embed some assumptions that were more appropriate to SMBus (and PCs) than to I2C. One of these assumptions was that most adapters and devices drivers support the SMBUS_QUICK protocol to probe device presence. Another was that devices and their drivers can be sufficiently configured using only such probe primitives.

As Linux and its I2C stack became more widely used in embedded systems and complex components such as DVB adapters, those assumptions became more problematic. Drivers for I2C devices that issue interrupts need more (and different) configuration information, as do drivers handling chip variants that can't be distinguished by protocol probing, or which need some board specific information to operate correctly.

Device/Driver Binding

System infrastructure, typically board-specific initialization code or boot firmware, reports what I2C devices exist. For example, there may be a table, in the kernel or from the boot loader, identifying I2C devices and linking them to board-specific configuration information about IRQs and other wiring artifacts, chip type, and so on. That could be used to create i2c_client objects for each I2C device.

I2C device drivers using this binding model work just like any other kind of driver in Linux: they provide a probe() method to bind to those devices, and a remove() method to unbind.

```
static int foo_probe(struct i2c_client *client,
                     const struct i2c_device_id *id);
static int foo_remove(struct i2c_client *client);
```

Remember that the i2c_driver does not create those client handles. The handle may be used during foo_probe(). If foo_probe() reports success (zero not a negative status code) it may save the handle and use it until foo_remove() returns. That binding model is used by most Linux drivers.

The probe function is called when an entry in the id_table name field matches the device's name. It is passed the entry that was matched so the driver knows which one in the table matched.

Device Creation

If you know for a fact that an I2C device is connected to a given I2C bus, you can instantiate that device by simply filling an i2c_board_info structure with the device address and driver name, and calling i2c_new_client_device(). This will create the device, then the driver core will take care of finding the right driver and will call its probe() method. If a driver supports different device types, you can specify the type you want using the type field. You can also specify an IRQ and platform data if needed.

Sometimes you know that a device is connected to a given I2C bus, but you don't know the exact address it uses. This happens on TV adapters for example, where the same driver supports dozens of slightly different models, and I2C device addresses change from one model to the next. In that case, you can use the i2c_new_scanned_device() variant, which is similar to i2c_new_client_device(), except that it takes an additional list of possible I2C addresses to probe. A device is created for the first responsive address in the list. If you expect more than one device to be present in the address range, simply call i2c_new_scanned_device() that many times.

The call to i2c_new_client_device() or i2c_new_scanned_device() typically happens in the I2C bus driver. You may want to save the returned i2c_client reference for later use.

Device Detection

Sometimes you do not know in advance which I2C devices are connected to a given I2C bus. This is for example the case of hardware monitoring devices on a PC's SMBus. In that case, you may want to let your driver detect supported devices automatically. This is how the legacy model was working, and is now available as an extension to the standard driver model.

You simply have to define a detect callback which will attempt to identify supported devices (returning 0 for supported ones and -ENODEV for unsupported ones), a list of addresses to probe, and a device type (or class) so that only I2C buses which may have that type of device connected (and not otherwise enumerated) will be probed. For example, a driver for a hardware monitoring chip for which auto-detection is needed would set its class to I2C_CLASS_HWMON, and only I2C adapters with a class including I2C_CLASS_HWMON would be probed by this driver. Note that the absence of matching classes does not prevent the use of a device of that type on the given I2C adapter. All it prevents is auto-detection; explicit instantiation of devices is still possible.

Note that this mechanism is purely optional and not suitable for all devices. You need some reliable way to identify the supported devices (typically using device-specific, dedicated identification registers), otherwise misdetections are likely to occur and things can get wrong quickly. Keep in mind that the I2C protocol doesn't include any standard way to detect the presence of a chip at a given address, let alone a standard way to identify devices. Even worse is the lack of semantics associated to bus transfers, which means that the same transfer can be seen as a read operation by a chip and as a write operation by another chip. For these reasons, explicit device instantiation should always be preferred to auto-detection where possible.

Device Deletion

Each I2C device which has been created using i2c_new_client_device() or i2c_new_scanned_device() can be unregistered by calling i2c_unregister_device(). If you don't call it explicitly, it will be called automatically before the underlying I2C bus itself is removed, as a device can't survive its parent in the device driver model.

2.1.6 Initializing the driver

When the kernel is booted, or when your foo driver module is inserted, you have to do some initializing. Fortunately, just registering the driver module is usually enough.

```
static int __init foo_init(void)
{
      return i2c_add_driver(&foo_driver);
}
module_init(foo_init);

static void __exit foo_cleanup(void)
{
      i2c_del_driver(&foo_driver);
}
module_exit(foo_cleanup);

The module_i2c_driver() macro can be used to reduce above code.

module_i2c_driver(foo_driver);
```

Note that some functions are marked by `__init`. These functions can be removed after kernel booting (or module loading) is completed. Likewise, functions marked by `__exit` are dropped by the compiler when the code is built into the kernel, as they would never be called.

2.1.7 Driver Information

```
/* Substitute your own name and email address */
MODULE_AUTHOR("Frodo Looijaard <frodol@dds.nl>"
MODULE_DESCRIPTION("Driver for Barf Inc. Foo I2C devices");

/* a few non-GPL license types are also allowed */
MODULE_LICENSE("GPL");
```

2.1.8 Power Management

If your I2C device needs special handling when entering a system low power state – like putting a transceiver into a low power mode, or activating a system wakeup mechanism – do that by implementing the appropriate callbacks for the dev_pm_ops of the driver (like suspend and resume).

These are standard driver model calls, and they work just like they would for any other driver stack. The calls can sleep, and can use I2C messaging to the device being suspended or resumed (since their parent I2C adapter is active when these calls are issued, and IRQs are still enabled).

2.1.9 System Shutdown

If your I2C device needs special handling when the system shuts down or reboots (including kexec) – like turning something off – use a shutdown() method.

Again, this is a standard driver model call, working just like it would for any other driver stack: the calls can sleep, and can use I2C messaging.

2.1.10 Command function

A generic ioctl-like function call back is supported. You will seldom need this, and its use is deprecated anyway, so newer design should not use it.

2.1.11 Sending and receiving

If you want to communicate with your device, there are several functions to do this. You can find all of them in <linux/i2c.h>.

If you can choose between plain I2C communication and SMBus level communication, please use the latter. All adapters understand SMBus level commands, but only some of them understand plain I2C!

Plain I2C communication

```
int i2c_master_send(struct i2c_client *client, const char *buf,
                    int count);
int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```

These routines read and write some bytes from/to a client. The client contains the I2C address, so you do not have to include it. The second parameter contains the bytes to read/write, the third the number of bytes to read/write (must be less than the length of the buffer, also should be less than 64k since msg.len is u16.) Returned is the actual number of bytes read/written.

```
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg,
                 int num);
```

This sends a series of messages. Each message can be a read or write, and they can be mixed in any way. The transactions are combined: no stop condition is issued between transaction. The i2c_msg structure contains for each message the client address, the number of bytes of the message and the message data itself.

You can read the file `i2c-protocol` for more information about the actual I2C protocol.

SMBus communication

```
s32 i2c_smbus_xfer(struct i2c_adapter *adapter, u16 addr,
                   unsigned short flags, char read_write, u8 command,
                   int size, union i2c_smbus_data *data);
```

This is the generic SMBus function. All functions below are implemented in terms of it. Never use this function directly!

```
s32 i2c_smbus_read_byte(struct i2c_client *client);
s32 i2c_smbus_write_byte(struct i2c_client *client, u8 value);
s32 i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
s32 i2c_smbus_write_byte_data(struct i2c_client *client,
                              u8 command, u8 value);
s32 i2c_smbus_read_word_data(struct i2c_client *client, u8 command);
s32 i2c_smbus_write_word_data(struct i2c_client *client,
                              u8 command, u16 value);
s32 i2c_smbus_read_block_data(struct i2c_client *client,
                              u8 command, u8 *values);
s32 i2c_smbus_write_block_data(struct i2c_client *client,
                               u8 command, u8 length, const u8 *values);
s32 i2c_smbus_read_i2c_block_data(struct i2c_client *client,
                                  u8 command, u8 length, u8 *values);
s32 i2c_smbus_write_i2c_block_data(struct i2c_client *client,
                                   u8 command, u8 length,
                                   const u8 *values);
```

These ones were removed from i2c-core because they had no users, but could be added back later if needed:

```
s32 i2c_smbus_write_quick(struct i2c_client *client, u8 value);
s32 i2c_smbus_process_call(struct i2c_client *client,
                           u8 command, u16 value);
s32 i2c_smbus_block_process_call(struct i2c_client *client,
                                 u8 command, u8 length, u8 *values);
```

All these transactions return a negative errno value on failure. The 'write' transactions return 0 on success; the 'read' transactions return the read value, except for block transactions, which return the number of values read. The block buffers need not be longer than 32 bytes.

You can read the file `smbus-protocol` for more information about the actual SMBus protocol.

2.1.12 General purpose routines

Below all general purpose routines are listed, that were not mentioned before:

```
/* Return the adapter number for a specific adapter */
int i2c_adapter_id(struct i2c_adapter *adap);
```

2.2 Implementing I2C device drivers in userspace

Usually, I2C devices are controlled by a kernel driver. But it is also possible to access all devices on an adapter from userspace, through the /dev interface. You need to load module i2c-dev for this.

Each registered I2C adapter gets a number, counting from 0. You can examine /sys/class/i2c-dev/ to see what number corresponds to which adapter. Alternatively, you can run "i2cdetect -l" to obtain a formatted list of all I2C adapters present on your system at a given time. i2cdetect is part of the i2c-tools package.

I2C device files are character device files with major device number 89 and a minor device number corresponding to the number assigned as explained above. They should be called "i2c-%d" (i2c-0, i2c-1, ⋯, i2c-10, ⋯). All 256 minor device numbers are reserved for I2C.

2.2.1 C example

So let's say you want to access an I2C adapter from a C program. First, you need to include these two headers:

```
#include <linux/i2c-dev.h>
#include <i2c/smbus.h>
```

Now, you have to decide which adapter you want to access. You should inspect /sys/class/i2c-dev/ or run "i2cdetect -l" to decide this. Adapter numbers are assigned somewhat dynamically, so you can not assume much about them. They can even change from one boot to the next.

Next thing, open the device file, as follows:

```
int file;
int adapter_nr = 2; /* probably dynamically determined */
char filename[20];

snprintf(filename, 19, "/dev/i2c-%d", adapter_nr);
file = open(filename, O_RDWR);
if (file < 0) {
  /* ERROR HANDLING; you can check errno to see what went wrong */
  exit(1);
}
```

When you have opened the device, you must specify with what device address you want to communicate:

```
int addr = 0x40; /* The I2C address */

if (ioctl(file, I2C_SLAVE, addr) < 0) {
  /* ERROR HANDLING; you can check errno to see what went wrong */
  exit(1);
}
```

Well, you are all set up now. You can now use SMBus commands or plain I2C to communicate with your device. SMBus commands are preferred if the device supports them. Both are illustrated below:

```
__u8 reg = 0x10; /* Device register to access */
__s32 res;
char buf[10];

/* Using SMBus commands */
res = i2c_smbus_read_word_data(file, reg);
if (res < 0) {
  /* ERROR HANDLING: I2C transaction failed */
} else {
  /* res contains the read word */
}

/*
 * Using I2C Write, equivalent of
 * i2c_smbus_write_word_data(file, reg, 0x6543)
 */
buf[0] = reg;
buf[1] = 0x43;
buf[2] = 0x65;
if (write(file, buf, 3) != 3) {
  /* ERROR HANDLING: I2C transaction failed */
}

/* Using I2C Read, equivalent of i2c_smbus_read_byte(file) */
if (read(file, buf, 1) != 1) {
  /* ERROR HANDLING: I2C transaction failed */
} else {
  /* buf[0] contains the read byte */
}
```

Note that only a subset of the I2C and SMBus protocols can be achieved by the means of read() and write() calls. In particular, so-called combined transactions (mixing read and write messages in the same transaction) aren’t supported. For this reason, this interface is almost never used by user-space programs.

IMPORTANT: because of the use of inline functions, you have to use ‘-O’ or some variation when you compile your program!

2.2.2 Full interface description

The following IOCTLs are defined:

ioctl(file, I2C_SLAVE, long addr) Change slave address. The address is passed in the 7 lower bits of the argument (except for 10 bit addresses, passed in the 10 lower bits in this case).

ioctl(file, I2C_TENBIT, long select) Selects ten bit addresses if select not equals 0, selects normal 7 bit addresses if select equals 0. Default 0. This request is only valid if the adapter has I2C_FUNC_10BIT_ADDR.

ioctl(file, I2C_PEC, long select) Selects SMBus PEC (packet error checking) generation and verification if select not equals 0, disables if select equals 0. Default 0. Used only for SMBus transactions. This request only has an effect if the the adapter has I2C_FUNC_SMBUS_PEC; it is still safe if not, it just doesn't have any effect.

ioctl(file, I2C_FUNCS, unsigned long *funcs) Gets the adapter functionality and puts it in *funcs.

ioctl(file, I2C_RDWR, struct i2c_rdwr_ioctl_data *msgset) Do combined read/write transaction without stop in between. Only valid if the adapter has I2C_FUNC_I2C. The argument is a pointer to a:

```
struct i2c_rdwr_ioctl_data {
  struct i2c_msg *msgs;  /* ptr to array of simple messages */
  int nmsgs;             /* number of messages to exchange */
}
```

The msgs[] themselves contain further pointers into data buffers. The function will write or read data to or from that buffers depending on whether the I2C_M_RD flag is set in a particular message or not. The slave address and whether to use ten bit address mode has to be set in each message, overriding the values set with the above ioctl's.

ioctl(file, I2C_SMBUS, struct i2c_smbus_ioctl_data *args) If possible, use the provided i2c_smbus_* methods described below instead of issuing direct ioctls.

You can do plain I2C transactions by using read(2) and write(2) calls. You do not need to pass the address byte; instead, set it through ioctl I2C_SLAVE before you try to access the device.

You can do SMBus level transactions (see documentation file smbus-protocol for details) through the following functions:

```
__s32 i2c_smbus_write_quick(int file, __u8 value);
__s32 i2c_smbus_read_byte(int file);
__s32 i2c_smbus_write_byte(int file, __u8 value);
__s32 i2c_smbus_read_byte_data(int file, __u8 command);
__s32 i2c_smbus_write_byte_data(int file, __u8 command, __u8 value);
__s32 i2c_smbus_read_word_data(int file, __u8 command);
__s32 i2c_smbus_write_word_data(int file, __u8 command, __u16 value);
__s32 i2c_smbus_process_call(int file, __u8 command, __u16 value);
__s32 i2c_smbus_read_block_data(int file, __u8 command, __u8 *values);
```

(continues on next page)

(continued from previous page)

```
__s32 i2c_smbus_write_block_data(int file, __u8 command, __u8 length,
                                 __u8 *values);
```

All these transactions return -1 on failure; you can read errno to see what happened. The 'write' transactions return 0 on success; the 'read' transactions return the read value, except for read_block, which returns the number of values read. The block buffers need not be longer than 32 bytes.

The above functions are made available by linking against the libi2c library, which is provided by the i2c-tools project. See: https://git.kernel.org/pub/scm/utils/i2c-tools/i2c-tools.git/.

2.2.3 Implementation details

For the interested, here's the code flow which happens inside the kernel when you use the /dev interface to I2C:

1) Your program opens /dev/i2c-N and calls ioctl() on it, as described in section "C example" above.

2) These open() and ioctl() calls are handled by the i2c-dev kernel driver: see i2c-dev.c:i2cdev_open() and i2c-dev.c:i2cdev_ioctl(), respectively. You can think of i2c-dev as a generic I2C chip driver that can be programmed from user-space.

3) Some ioctl() calls are for administrative tasks and are handled by i2c-dev directly. Examples include I2C_SLAVE (set the address of the device you want to access) and I2C_PEC (enable or disable SMBus error checking on future transactions.)

4) Other ioctl() calls are converted to in-kernel function calls by i2c-dev. Examples include I2C_FUNCS, which queries the I2C adapter functionality using i2c.h:i2c_get_functionality(), and I2C_SMBUS, which performs an SMBus transaction using i2c-core-smbus.c:i2c_smbus_xfer().

   The i2c-dev driver is responsible for checking all the parameters that come from user-space for validity. After this point, there is no difference between these calls that came from user-space through i2c-dev and calls that would have been performed by kernel I2C chip drivers directly. This means that I2C bus drivers don't need to implement anything special to support access from user-space.

5) These i2c.h functions are wrappers to the actual implementation of your I2C bus driver. Each adapter must declare callback functions implementing these standard calls. i2c.h:i2c_get_functionality() calls i2c_adapter.algo->functionality(), while i2c-core-smbus.c:i2c_smbus_xfer() calls either adapter.algo->smbus_xfer() if it is implemented, or if not, i2c-core-smbus.c:i2c_smbus_xfer_emulated() which in turn calls i2c_adapter.algo->master_xfer().

After your I2C bus driver has processed these requests, execution runs up the call chain, with almost no processing done, except by i2c-dev to package the returned data, if any, in suitable format for the ioctl.

2.3 Linux I2C and DMA

Given that I2C is a low-speed bus, over which the majority of messages transferred are small, it is not considered a prime user of DMA access. At this time of writing, only 10% of I2C bus master drivers have DMA support implemented. And the vast majority of transactions are so small that setting up DMA for it will likely add more overhead than a plain PIO transfer.

Therefore, it is not mandatory that the buffer of an I2C message is DMA safe. It does not seem reasonable to apply additional burdens when the feature is so rarely used. However, it is recommended to use a DMA-safe buffer if your message size is likely applicable for DMA. Most drivers have this threshold around 8 bytes (as of today, this is mostly an educated guess, however). For any message of 16 byte or larger, it is probably a really good idea. Please note that other subsystems you use might add requirements. E.g., if your I2C bus master driver is using USB as a bridge, then you need to have DMA safe buffers always, because USB requires it.

2.3.1 Clients

For clients, if you use a DMA safe buffer in i2c_msg, set the I2C_M_DMA_SAFE flag with it. Then, the I2C core and drivers know they can safely operate DMA on it. Note that using this flag is optional. I2C host drivers which are not updated to use this flag will work like before. And like before, they risk using an unsafe DMA buffer. To improve this situation, using I2C_M_DMA_SAFE in more and more clients and host drivers is the planned way forward. Note also that setting this flag makes only sense in kernel space. User space data is copied into kernel space anyhow. The I2C core makes sure the destination buffers in kernel space are always DMA capable. Also, when the core emulates SMBus transactions via I2C, the buffers for block transfers are DMA safe. Users of i2c_master_send() and i2c_master_recv() functions can now use DMA safe variants (i2c_master_send_dmasafe() and i2c_master_recv_dmasafe()) once they know their buffers are DMA safe. Users of i2c_transfer() must set the I2C_M_DMA_SAFE flag manually.

2.3.2 Masters

Bus master drivers wishing to implement safe DMA can use helper functions from the I2C core. One gives you a DMA-safe buffer for a given i2c_msg as long as a certain threshold is met:

```
dma_buf = i2c_get_dma_safe_msg_buf(msg, threshold_in_byte);
```

If a buffer is returned, it is either msg->buf for the I2C_M_DMA_SAFE case or a bounce buffer. But you don’ t need to care about that detail, just use the returned buffer. If NULL is returned, the threshold was not met or a bounce buffer could not be allocated. Fall back to PIO in that case.

In any case, a buffer obtained from above needs to be released. Another helper function ensures a potentially used bounce buffer is freed:

```
i2c_put_dma_safe_msg_buf(dma_buf, msg, xferred);
```

The last argument ‘xferred’ controls if the buffer is synced back to the message or not. No syncing is needed in cases setting up DMA had an error and there was no data transferred.

The bounce buffer handling from the core is generic and simple. It will always allocate a new bounce buffer. If you want a more sophisticated handling (e.g. reusing pre-allocated buffers), you are free to implement your own.

Please also check the in-kernel documentation for details. The i2c-sh_mobile driver can be used as a reference example how to use the above helpers.

Final note: If you plan to use DMA with I2C (or with anything else, actually) make sure you have CONFIG_DMA_API_DEBUG enabled during development. It can help you find various issues which can be complex to debug otherwise.

2.4 I2C/SMBUS Fault Codes

This is a summary of the most important conventions for use of fault codes in the I2C/SMBus stack.

2.4.1 A “Fault” is not always an “Error”

Not all fault reports imply errors; “page faults” should be a familiar example. Software often retries idempotent operations after transient faults. There may be fancier recovery schemes that are appropriate in some cases, such as re-initializing (and maybe resetting). After such recovery, triggered by a fault report, there is no error.

In a similar way, sometimes a “fault” code just reports one defined result for an operation ···it doesn’t indicate that anything is wrong at all, just that the outcome wasn’t on the “golden path”.

In short, your I2C driver code may need to know these codes in order to respond correctly. Other code may need to rely on YOUR code reporting the right fault code, so that it can (in turn) behave correctly.

2.4.2 I2C and SMBus fault codes

These are returned as negative numbers from most calls, with zero or some positive number indicating a non-fault return. The specific numbers associated with these symbols differ between architectures, though most Linux systems use <asm-generic/errno*.h> numbering.

Note that the descriptions here are not exhaustive. There are other codes that may be returned, and other cases where these codes should be returned. However, drivers should not return other codes for these cases (unless the hardware doesn’t provide unique fault reports).

Also, codes returned by adapter probe methods follow rules which are specific to their host bus (such as PCI, or the platform bus).

EAGAIN Returned by I2C adapters when they lose arbitration in master transmit mode: some other master was transmitting different data at the same time.

Also returned when trying to invoke an I2C operation in an atomic context, when some task is already using that I2C bus to execute some other operation.

EBADMSG Returned by SMBus logic when an invalid Packet Error Code byte is received. This code is a CRC covering all bytes in the transaction, and is sent before the terminating STOP. This fault is only reported on read transactions; the SMBus slave may have a way to report PEC mismatches on writes from the host. Note that even if PECs are in use, you should not rely on these as the only way to detect incorrect data transfers.

EBUSY Returned by SMBus adapters when the bus was busy for longer than allowed. This usually indicates some device (maybe the SMBus adapter) needs some fault recovery (such as resetting), or that the reset was attempted but failed.

EINVAL This rather vague error means an invalid parameter has been detected before any I/O operation was started. Use a more specific fault code when you can.

EIO This rather vague error means something went wrong when performing an I/O operation. Use a more specific fault code when you can.

ENODEV Returned by driver probe() methods. This is a bit more specific than ENXIO, implying the problem isn't with the address, but with the device found there. Driver probes may verify the device returns correct responses, and return this as appropriate. (The driver core will warn about probe faults other than ENXIO and ENODEV.)

ENOMEM Returned by any component that can't allocate memory when it needs to do so.

ENXIO Returned by I2C adapters to indicate that the address phase of a transfer didn't get an ACK. While it might just mean an I2C device was temporarily not responding, usually it means there's nothing listening at that address.

Returned by driver probe() methods to indicate that they found no device to bind to. (ENODEV may also be used.)

EOPNOTSUPP Returned by an adapter when asked to perform an operation that it doesn't, or can't, support.

For example, this would be returned when an adapter that doesn't support SMBus block transfers is asked to execute one. In that case, the driver making that request should have verified that functionality was supported before it made that block transfer request.

Similarly, if an I2C adapter can't execute all legal I2C messages, it should return this when asked to perform a transaction it can't. (These limitations can't be seen in the adapter's functionality mask, since the assumption is that if an adapter supports I2C it supports all of I2C.)

EPROTO Returned when slave does not conform to the relevant I2C or SMBus (or chip-specific) protocol specifications. One case is when the length of an SMBus block data response (from the SMBus slave) is outside the range 1-32 bytes.

ESHUTDOWN Returned when a transfer was requested using an adapter which is already suspended.

ETIMEDOUT This is returned by drivers when an operation took too much time, and was aborted before it completed.

SMBus adapters may return it when an operation took more time than allowed by the SMBus specification; for example, when a slave stretches clocks too far. I2C has no such timeouts, but it's normal for I2C adapters to impose some arbitrary limits (much longer than SMBus!) too.

2.5 I2C/SMBus Functionality

2.5.1 INTRODUCTION

Because not every I2C or SMBus adapter implements everything in the I2C specifications, a client can not trust that everything it needs is implemented when it is given the option to attach to an adapter: the client needs some way to check whether an adapter has the needed functionality.

2.5.2 FUNCTIONALITY CONSTANTS

For the most up-to-date list of functionality constants, please check <uapi/linux/i2c.h>!

I2C_FUNC_I2C	Plain i2c-level commands (Pure SMBus adapters typically can not do these)
I2C_FUNC_10BIT_ADDR	Handles the 10-bit address extensions
I2C_FUNC_PROTOCOL_MANGLING	Knows about the I2C_M_IGNORE_NAK, I2C_M_REV_DIR_ADDR and I2C_M_NO_RD_ACK flags (which modify the I2C protocol!)
I2C_FUNC_NOSTART	Can skip repeated start sequence
I2C_FUNC_SMBUS_QUICK	Handles the SMBus write_quick command
I2C_FUNC_SMBUS_READ_BYTE	Handles the SMBus read_byte command
I2C_FUNC_SMBUS_WRITE_BYTE	Handles the SMBus write_byte command
I2C_FUNC_SMBUS_READ_BYTE_DATA	Handles the SMBus read_byte_data command
I2C_FUNC_SMBUS_WRITE_BYTE_DATA	Handles the SMBus write_byte_data command
I2C_FUNC_SMBUS_READ_WORD_DATA	Handles the SMBus read_word_data command
I2C_FUNC_SMBUS_WRITE_WORD_DATA	Handles the SMBus write_byte_data command
I2C_FUNC_SMBUS_PROC_CALL	Handles the SMBus process_call command
I2C_FUNC_SMBUS_READ_BLOCK_DATA	Handles the SMBus read_block_data command
I2C_FUNC_SMBUS_WRITE_BLOCK_DATA	Handles the SMBus write_block_data command
I2C_FUNC_SMBUS_READ_I2C_BLOCK	Handles the SMBus read_i2c_block_data command
I2C_FUNC_SMBUS_WRITE_I2C_BLOCK	Handles the SMBus write_i2c_block_data command

A few combinations of the above flags are also defined for your convenience:

I2C_FUNC_SMBUS_BYTE	Handles the SMBus read_byte and write_byte commands
I2C_FUNC_SMBUS_BYTE_DATA	Handles the SMBus read_byte_data and write_byte_data commands
I2C_FUNC_SMBUS_WORD_DATA	Handles the SMBus read_word_data and write_word_data commands
I2C_FUNC_SMBUS_BLOCK_DATA	Handles the SMBus read_block_data and write_block_data commands
I2C_FUNC_SMBUS_I2C_BLOCK	Handles the SMBus read_i2c_block_data and write_i2c_block_data commands
I2C_FUNC_SMBUS_EMUL	Handles all SMBus commands that can be emulated by a real I2C adapter (using the transparent emulation layer)

In kernel versions prior to 3.5 I2C_FUNC_NOSTART was implemented as part of I2C_FUNC_PROTOCOL_MANGLING.

2.5.3 ADAPTER IMPLEMENTATION

When you write a new adapter driver, you will have to implement a function callback `functionality`. Typical implementations are given below.

A typical SMBus-only adapter would list all the SMBus transactions it supports. This example comes from the i2c-piix4 driver:

```
static u32 piix4_func(struct i2c_adapter *adapter)
{
      return I2C_FUNC_SMBUS_QUICK | I2C_FUNC_SMBUS_BYTE |
             I2C_FUNC_SMBUS_BYTE_DATA | I2C_FUNC_SMBUS_WORD_DATA |
             I2C_FUNC_SMBUS_BLOCK_DATA;
}
```

A typical full-I2C adapter would use the following (from the i2c-pxa driver):

```
static u32 i2c_pxa_functionality(struct i2c_adapter *adap)
{
      return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL;
}
```

I2C_FUNC_SMBUS_EMUL includes all the SMBus transactions (with the addition of I2C block transactions) which i2c-core can emulate using I2C_FUNC_I2C without any help from the adapter driver. The idea is to let the client drivers check for the support of SMBus functions without having to care whether the said functions are implemented in hardware by the adapter, or emulated in software by i2c-core on top of an I2C adapter.

2.5.4 CLIENT CHECKING

Before a client tries to attach to an adapter, or even do tests to check whether one of the devices it supports is present on an adapter, it should check whether the needed functionality is present. The typical way to do this is (from the lm75 driver):

```
static int lm75_detect(...)
{
      (...)
      if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_BYTE_DATA |
                                   I2C_FUNC_SMBUS_WORD_DATA))
              goto exit;
      (...)
}
```

Here, the lm75 driver checks if the adapter can do both SMBus byte data and SMBus word data transactions. If not, then the driver won't work on this adapter and there's no point in going on. If the check above is successful, then the driver knows that it can call the following functions: i2c_smbus_read_byte_data(), i2c_smbus_write_byte_data(), i2c_smbus_read_word_data() and i2c_smbus_write_word_data(). As a rule of thumb, the functionality constants you test for with i2c_check_functionality() should match exactly the i2c_smbus_* functions which you driver is calling.

Note that the check above doesn't tell whether the functionalities are implemented in hardware by the underlying adapter or emulated in software by i2c-core. Client drivers don't have to care about this, as i2c-core will transparently implement SMBus transactions on top of I2C adapters.

2.5.5 CHECKING THROUGH /DEV

If you try to access an adapter from a userspace program, you will have to use the /dev interface. You will still have to check whether the functionality you need is supported, of course. This is done using the I2C_FUNCS ioctl. An example, adapted from the i2cdetect program, is below:

```
int file;
if (file = open("/dev/i2c-0", O_RDWR) < 0) {
      /* Some kind of error handling */
      exit(1);
}
if (ioctl(file, I2C_FUNCS, &funcs) < 0) {
      /* Some kind of error handling */
      exit(1);
}
if (!(funcs & I2C_FUNC_SMBUS_QUICK)) {
      /* Oops, the needed functionality (SMBus write_quick function) is
         not available! */
      exit(1);
}
/* Now it is safe to use the SMBus write_quick command */
```

CHAPTER

THREE

DEBUGGING

3.1 Linux I2C fault injection

The GPIO based I2C bus master driver can be configured to provide fault injection capabilities. It is then meant to be connected to another I2C bus which is driven by the I2C bus master driver under test. The GPIO fault injection driver can create special states on the bus which the other I2C bus master driver should handle gracefully.

Once the Kconfig option I2C_GPIO_FAULT_INJECTOR is enabled, there will be an 'i2c-fault-injector' subdirectory in the Kernel debugfs filesystem, usually mounted at /sys/kernel/debug. There will be a separate subdirectory per GPIO driven I2C bus. Each subdirectory will contain files to trigger the fault injection. They will be described now along with their intended use-cases.

3.1.1 Wire states

"scl"

By reading this file, you get the current state of SCL. By writing, you can change its state to either force it low or to release it again. So, by using "echo 0 > scl" you force SCL low and thus, no communication will be possible because the bus master under test will not be able to clock. It should detect the condition of SCL being unresponsive and report an error to the upper layers.

"sda"

By reading this file, you get the current state of SDA. By writing, you can change its state to either force it low or to release it again. So, by using "echo 0 > sda" you force SDA low and thus, data cannot be transmitted. The bus master under test should detect this condition and trigger a bus recovery (see I2C specification version 4, section 3.1.16) using the helpers of the Linux I2C core (see 'struct bus_recovery_info'). However, the bus recovery will not succeed because SDA is still pinned low until you manually release it again with "echo 1 > sda". A test with an automatic release can be done with the "incomplete transfers" class of fault injectors.

3.1.2 Incomplete transfers

The following fault injectors create situations where SDA will be held low by a device. Bus recovery should be able to fix these situations. But please note: there are I2C client devices which detect a stuck SDA on their side and release it on their own after a few milliseconds. Also, there might be an external device deglitching and monitoring the I2C bus. It could also detect a stuck SDA and will init a bus recovery on its own. If you want to implement bus recovery in a bus master driver, make sure you checked your hardware setup for such devices before. And always verify with a scope or logic analyzer!

“incomplete_address_phase”

This file is write only and you need to write the address of an existing I2C client device to it. Then, a read transfer to this device will be started, but it will stop at the ACK phase after the address of the client has been transmitted. Because the device will ACK its presence, this results in SDA being pulled low by the device while SCL is high. So, similar to the “sda” file above, the bus master under test should detect this condition and try a bus recovery. This time, however, it should succeed and the device should release SDA after toggling SCL.

“incomplete_write_byte”

Similar to above, this file is write only and you need to write the address of an existing I2C client device to it.

The injector will again stop at one ACK phase, so the device will keep SDA low because it acknowledges data. However, there are two differences compared to ‘incomplete_address_phase’ :

a) the message sent out will be a write message

b) after the address byte, a 0x00 byte will be transferred. Then, stop at ACK.

This is a highly delicate state, the device is set up to write any data to register 0x00 (if it has registers) when further clock pulses happen on SCL. This is why bus recovery (up to 9 clock pulses) must either check SDA or send additional STOP conditions to ensure the bus has been released. Otherwise random data will be written to a device!

3.1.3 Lost arbitration

Here, we want to simulate the condition where the master under test loses the bus arbitration against another master in a multi-master setup.

“lose_arbitration”

This file is write only and you need to write the duration of the arbitration interference (in µs, maximum is 100ms). The calling process will then sleep and wait for the next bus clock. The process is interruptible, though.

Arbitration lost is achieved by waiting for SCL going down by the master under test and then pulling SDA low for some time. So, the I2C address sent out should be corrupted and that should be detected properly. That means that the address sent out should have a lot of ‘1’ bits to be able to detect corruption. There doesn’t need to be a device at this address because arbitration lost should be detected beforehand. Also note, that SCL going down is monitored using interrupts, so the interrupt latency might cause the first bits to be not corrupted. A good starting point for using this fault injector on an otherwise idle bus is:

```
# echo 200 > lose_arbitration &
# i2cget -y <bus_to_test> 0x3f
```

3.1.4 Panic during transfer

This fault injector will create a Kernel panic once the master under test started a transfer. This usually means that the state machine of the bus master driver will be ungracefully interrupted and the bus may end up in an unusual state. Use this to check if your shutdown/reboot/boot code can handle this scenario.

“inject_panic”

This file is write only and you need to write the delay between the detected start of a transmission and the induced Kernel panic (in µs, maximum is 100ms). The calling process will then sleep and wait for the next bus clock. The process is interruptible, though.

Start of a transfer is detected by waiting for SCL going down by the master under test. A good starting point for using this fault injector is:

```
# echo 0 > inject_panic &
# i2cget -y <bus_to_test> <some_address>
```

Note that there doesn’t need to be a device listening to the address you are using. Results may vary depending on that, though.

3.2 i2c-stub

3.2.1 Description

This module is a very simple fake I2C/SMBus driver. It implements six types of SMBus commands: write quick, (r/w) byte, (r/w) byte data, (r/w) word data, (r/w) I2C block data, and (r/w) SMBus block data.

You need to provide chip addresses as a module parameter when loading this driver, which will then only react to SMBus commands to these addresses.

No hardware is needed nor associated with this module. It will accept write quick commands to the specified addresses; it will respond to the other commands (also to the specified addresses) by reading from or writing to arrays in memory. It will also spam the kernel logs for every command it handles.

A pointer register with auto-increment is implemented for all byte operations. This allows for continuous byte reads like those supported by EEPROMs, among others.

SMBus block command support is disabled by default, and must be enabled explicitly by setting the respective bits (0x03000000) in the functionality module parameter.

SMBus block commands must be written to configure an SMBus command for SMBus block operations. Writes can be partial. Block read commands always return the number of bytes selected with the largest write so far.

The typical use-case is like this:

1. load this module
2. use i2cset (from the i2c-tools project) to pre-load some data
3. load the target chip driver module
4. observe its behavior in the kernel log

There's a script named i2c-stub-from-dump in the i2c-tools package which can load register values automatically from a chip dump.

3.2.2 Parameters

int chip_addr[10]: The SMBus addresses to emulate chips at.

unsigned long functionality: Functionality override, to disable some commands. See I2C_FUNC_* constants in <linux/i2c.h> for the suitable values. For example, value 0x1f0000 would only enable the quick, byte and byte data commands.

u8 bank_reg[10], u8 bank_mask[10], u8 bank_start[10], u8 bank_end[10]: Optional bank settings. They tell which bits in which register select the active bank, as well as the range of banked registers.

3.2.3 Caveats

If your target driver polls some byte or word waiting for it to change, the stub could lock it up. Use i2cset to unlock it.

If you spam it hard enough, printk can be lossy. This module really wants something like relayfs.

CHAPTER

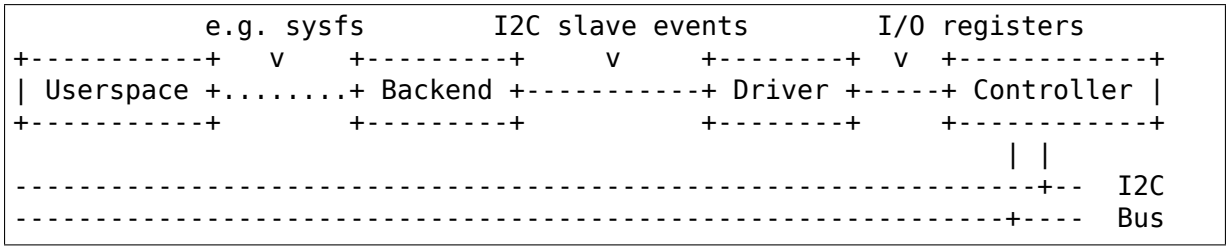
FOUR

SLAVE I2C

4.1 Linux I2C slave interface description

by Wolfram Sang <wsa@sang-engineering.com> in 2014-15

Linux can also be an I2C slave if the I2C controller in use has slave functionality. For that to work, one needs slave support in the bus driver plus a hardware independent software backend providing the actual functionality. An example for the latter is the slave-eeprom driver, which acts as a dual memory driver. While another I2C master on the bus can access it like a regular EEPROM, the Linux I2C slave can access the content via sysfs and handle data as needed. The backend driver and the I2C bus driver communicate via events. Here is a small graph visualizing the data flow and the means by which data is transported. The dotted line marks only one example. The backend could also use a character device, be in-kernel only, or something completely different:



Note: Technically, there is also the I2C core between the backend and the driver. However, at this time of writing, the layer is transparent.

4.1.1 User manual

I2C slave backends behave like standard I2C clients. So, you can instantiate them as described in the document 'instantiating-devices'. The only difference is that i2c slave backends have their own address space. So, you have to add 0x1000 to the address you would originally request. An example for instantiating the slave-eeprom driver from userspace at the 7 bit address 0x64 on bus 1:

```
# echo slave-24c02 0x1064 > /sys/bus/i2c/devices/i2c-1/new_device
```

Each backend should come with separate documentation to describe its specific behaviour and setup.

4.1.2 Developer manual

First, the events which are used by the bus driver and the backend will be described in detail. After that, some implementation hints for extending bus drivers and writing backends will be given.

I2C slave events

The bus driver sends an event to the backend using the following function:

```
ret = i2c_slave_event(client, event, &val)
```

'client' describes the I2C slave device. 'event' is one of the special event types described hereafter. 'val' holds an u8 value for the data byte to be read/written and is thus bidirectional. The pointer to val must always be provided even if val is not used for an event, i.e. don't use NULL here. 'ret' is the return value from the backend. Mandatory events must be provided by the bus drivers and must be checked for by backend drivers.

Event types:

- I2C_SLAVE_WRITE_REQUESTED (mandatory)

  'val' : unused

  'ret' : always 0

Another I2C master wants to write data to us. This event should be sent once our own address and the write bit was detected. The data did not arrive yet, so there is nothing to process or return. Wakeup or initialization probably needs to be done, though.

- I2C_SLAVE_READ_REQUESTED (mandatory)

  'val' : backend returns first byte to be sent

  'ret' : always 0

Another I2C master wants to read data from us. This event should be sent once our own address and the read bit was detected. After returning, the bus driver should transmit the first byte.

- I2C_SLAVE_WRITE_RECEIVED (mandatory)

  'val' : bus driver delivers received byte

  'ret' : 0 if the byte should be acked, some errno if the byte should be nacked

Another I2C master has sent a byte to us which needs to be set in 'val'. If 'ret' is zero, the bus driver should ack this byte. If 'ret' is an errno, then the byte should be nacked.

- I2C_SLAVE_READ_PROCESSED (mandatory)

  'val' : backend returns next byte to be sent

  'ret' : always 0

The bus driver requests the next byte to be sent to another I2C master in 'val'. Important: This does not mean that the previous byte has been acked, it only means that the previous byte is shifted out to the bus! To ensure seamless transmission, most hardware requests the next byte when the previous one is still shifted out. If the master sends NACK and stops reading after the byte currently shifted out, this byte requested here is never used. It very likely needs to be sent again on the next I2C_SLAVE_READ_REQUEST, depending a bit on your backend, though.

- I2C_SLAVE_STOP (mandatory)

  'val' : unused

  'ret' : always 0

A stop condition was received. This can happen anytime and the backend should reset its state machine for I2C transfers to be able to receive new requests.

Software backends

If you want to write a software backend:

- use a standard i2c_driver and its matching mechanisms
- write the slave_callback which handles the above slave events (best using a state machine)
- register this callback via i2c_slave_register()

Check the i2c-slave-eeprom driver as an example.

Bus driver support

If you want to add slave support to the bus driver:

- implement calls to register/unregister the slave and add those to the struct i2c_algorithm. When registering, you probably need to set the I2C slave address and enable slave specific interrupts. If you use runtime pm, you should use pm_runtime_get_sync() because your device usually needs to be powered on always to be able to detect its slave address. When unregistering, do the inverse of the above.
- Catch the slave interrupts and send appropriate i2c_slave_events to the backend.

Note that most hardware supports being master _and_ slave on the same bus. So, if you extend a bus driver, please make sure that the driver supports that as well. In almost all cases, slave support does not need to disable the master functionality.

Check the i2c-rcar driver as an example.

About ACK/NACK

It is good behaviour to always ACK the address phase, so the master knows if a device is basically present or if it mysteriously disappeared. Using NACK to state being busy is troublesome. SMBus demands to always ACK the address phase, while the I2C specification is more loose on that. Most I2C controllers also automatically ACK when detecting their slave addresses, so there is no option to NACK them. For those reasons, this API does not support NACK in the address phase.

Currently, there is no slave event to report if the master did ACK or NACK a byte when it reads from us. We could make this an optional event if the need arises. However, cases should be extremely rare because the master is expected to send STOP after that and we have an event for that. Also, keep in mind not all I2C controllers have the possibility to report that event.

About buffers

During development of this API, the question of using buffers instead of just bytes came up. Such an extension might be possible, usefulness is unclear at this time of writing. Some points to keep in mind when using buffers:

- Buffers should be opt-in and backend drivers will always have to support byte-based transactions as the ultimate fallback anyhow because this is how the majority of HW works.
- For backends simulating hardware registers, buffers are largely not helpful because after each byte written an action should be immediately triggered. For reads, the data kept in the buffer might get stale if the backend just updated a register because of internal processing.
- A master can send STOP at any time. For partially transferred buffers, this means additional code to handle this exception. Such code tends to be error-prone.

4.2 Linux I2C slave EEPROM backend

by Wolfram Sang <wsa@sang-engineering.com> in 2014-20

This backend simulates an EEPROM on the connected I2C bus. Its memory contents can be accessed from userspace via this file located in sysfs:

```
/sys/bus/i2c/devices/<device-directory>/slave-eeprom
```

The following types are available: 24c02, 24c32, 24c64, and 24c512. Read-only variants are also supported. The name needed for instantiating has the form 'slave-<type>[ro]'. Examples follow:

24c02, read/write, address 0x64: # echo slave-24c02 0x1064 > /sys/bus/i2c/devices/i2c-1/new_device

24c512, read-only, address 0x42: # echo slave-24c512ro 0x1042 > /sys/bus/i2c/devices/i2c-1/new_device

You can also preload data during boot if a device-property named ‘firmware-name’ contains a valid filename (DT or ACPI only).

As of 2015, Linux doesn’t support poll on binary sysfs files, so there is no notification when another master changed the content.

CHAPTER

FIVE

ADVANCED TOPICS

5.1 I2C Ten-bit Addresses

The I2C protocol knows about two kinds of device addresses: normal 7 bit addresses, and an extended set of 10 bit addresses. The sets of addresses do not intersect: the 7 bit address 0x10 is not the same as the 10 bit address 0x10 (though a single device could respond to both of them). To avoid ambiguity, the user sees 10 bit addresses mapped to a different address space, namely 0xa000-0xa3ff. The leading 0xa (= 10) represents the 10 bit mode. This is used for creating device names in sysfs. It is also needed when instantiating 10 bit devices via the new_device file in sysfs.

I2C messages to and from 10-bit address devices have a different format. See the I2C specification for the details.

The current 10 bit address support is minimal. It should work, however you can expect some problems along the way:

- Not all bus drivers support 10-bit addresses. Some don't because the hardware doesn't support them (SMBus doesn't require 10-bit address support for example), some don't because nobody bothered adding the code (or it's there but not working properly.) Software implementation (i2c-algo-bit) is known to work.
- Some optional features do not support 10-bit addresses. This is the case of automatic detection and instantiation of devices by their, drivers, for example.
- Many user-space packages (for example i2c-tools) lack support for 10-bit addresses.

Note that 10-bit address devices are still pretty rare, so the limitations listed above could stay for a long time, maybe even forever if nobody needs them to be fixed.

CHAPTER

SIX

LEGACY DOCUMENTATION

6.1 Upgrading I2C Drivers to the new 2.6 Driver Model

Ben Dooks <ben-linux@fluff.org>

6.1.1 Introduction

This guide outlines how to alter existing Linux 2.6 client drivers from the old to the new new binding methods.

6.1.2 Example old-style driver

```
struct example_state {
      struct i2c_client       client;
      ....
};

static struct i2c_driver example_driver;

static unsigned short ignore[] = { I2C_CLIENT_END };
static unsigned short normal_addr[] = { OUR_ADDR, I2C_CLIENT_END };

I2C_CLIENT_INSMOD;

static int example_attach(struct i2c_adapter *adap, int addr, int kind)
{
      struct example_state *state;
      struct device *dev = &adap->dev;  /* to use for dev_ reports */
      int ret;

      state = kzalloc(sizeof(struct example_state), GFP_KERNEL);
      if (state == NULL) {
              dev_err(dev, "failed to create our state\n");
              return -ENOMEM;
      }

      example->client.addr    = addr;
      example->client.flags   = 0;
      example->client.adapter = adap;

      i2c_set_clientdata(&state->i2c_client, state);
```

(continues on next page)

(continued from previous page)

```
      strscpy(client->i2c_client.name, "example", sizeof(client->i2c_
↪client.name));

      ret = i2c_attach_client(&state->i2c_client);
      if (ret < 0) {
              dev_err(dev, "failed to attach client\n");
              kfree(state);
              return ret;
      }

      dev = &state->i2c_client.dev;

      /* rest of the initialisation goes here. */

      dev_info(dev, "example client created\n");

      return 0;
}

static int example_detach(struct i2c_client *client)
{
      struct example_state *state = i2c_get_clientdata(client);

      i2c_detach_client(client);
      kfree(state);
      return 0;
}

static int example_attach_adapter(struct i2c_adapter *adap)
{
      return i2c_probe(adap, &addr_data, example_attach);
}

static struct i2c_driver example_driver = {
      .driver         = {
              .owner          = THIS_MODULE,
              .name           = "example",
              .pm             = &example_pm_ops,
      },
      .attach_adapter = example_attach_adapter,
      .detach_client  = example_detach,
};
```

6.1.3 Updating the client

The new style binding model will check against a list of supported devices and their associated address supplied by the code registering the busses. This means that the driver .attach_adapter and .detach_client methods can be removed, along with the addr_data, as follows:

```
- static struct i2c_driver example_driver;

- static unsigned short ignore[] = { I2C_CLIENT_END };
- static unsigned short normal_addr[] = { OUR_ADDR, I2C_CLIENT_END };
```

(continues on next page)

(continued from previous page)

```
- I2C_CLIENT_INSMOD;

- static int example_attach_adapter(struct i2c_adapter *adap)
- {
-     return i2c_probe(adap, &addr_data, example_attach);
- }

  static struct i2c_driver example_driver = {
-     .attach_adapter = example_attach_adapter,
-     .detach_client  = example_detach,
  }
```

Add the probe and remove methods to the i2c_driver, as so:

```
 static struct i2c_driver example_driver = {
+     .probe          = example_probe,
+     .remove         = example_remove,
 }
```

Change the example_attach method to accept the new parameters which include the i2c_client that it will be working with:

```
- static int example_attach(struct i2c_adapter *adap, int addr, int kind)
+ static int example_probe(struct i2c_client *client,
+                          const struct i2c_device_id *id)
```

Change the name of example_attach to example_probe to align it with the i2c_driver entry names. The rest of the probe routine will now need to be changed as the i2c_client has already been setup for use.

The necessary client fields have already been setup before the probe function is called, so the following client setup can be removed:

```
-     example->client.addr    = addr;
-     example->client.flags   = 0;
-     example->client.adapter = adap;
-
-     strscpy(client->i2c_client.name, "example", sizeof(client->i2c_
↪client.name));
```

The i2c_set_clientdata is now:

```
-     i2c_set_clientdata(&state->client, state);
+     i2c_set_clientdata(client, state);
```

The call to i2c_attach_client is no longer needed, if the probe routine exits successfully, then the driver will be automatically attached by the core. Change the probe routine as so:

```
-     ret = i2c_attach_client(&state->i2c_client);
-     if (ret < 0) {
-             dev_err(dev, "failed to attach client\n");
-             kfree(state);
```

(continues on next page)

(continued from previous page)

```
-             return ret;
-     }
```

Remove the storage of ‘struct i2c_client’ from the ‘struct example_state’ as we are provided with the i2c_client in our example_probe. Instead we store a pointer to it for when it is needed.

```
struct example_state {
-     struct i2c_client       client;
+     struct i2c_client       *client;
```

the new i2c client as so:

```
-     struct device *dev = &adap->dev;  /* to use for dev_ reports */
+     struct device *dev = &i2c_client->dev;  /* to use for dev_ reports */
```

And remove the change after our client is attached, as the driver no longer needs to register a new client structure with the core:

```
-     dev = &state->i2c_client.dev;
```

In the probe routine, ensure that the new state has the client stored in it:

```
static int example_probe(struct i2c_client *i2c_client,
                         const struct i2c_device_id *id)
{
      struct example_state *state;
      struct device *dev = &i2c_client->dev;
      int ret;

      state = kzalloc(sizeof(struct example_state), GFP_KERNEL);
      if (state == NULL) {
              dev_err(dev, "failed to create our state\n");
              return -ENOMEM;
      }

+     state->client = i2c_client;
```

Update the detach method, by changing the name to _remove and to delete the i2c_detach_client call. It is possible that you can also remove the ret variable as it is not needed for any of the core functions.

```
- static int example_detach(struct i2c_client *client)
+ static int example_remove(struct i2c_client *client)
{
      struct example_state *state = i2c_get_clientdata(client);

-     i2c_detach_client(client);
```

And finally ensure that we have the correct ID table for the i2c-core and other utilities:

```
+ struct i2c_device_id example_idtable[] = {
+       { "example", 0 },
```

(continues on next page)

(continued from previous page)

```
+       { }
+};
+
+MODULE_DEVICE_TABLE(i2c, example_idtable);

static struct i2c_driver example_driver = {
      .driver         = {
              .owner          = THIS_MODULE,
              .name           = "example",
      },
+     .id_table       = example_ids,
```

Our driver should now look like this:

```
struct example_state {
      struct i2c_client       *client;
      ....
};

static int example_probe(struct i2c_client *client,
                         const struct i2c_device_id *id)
{
      struct example_state *state;
      struct device *dev = &client->dev;

      state = kzalloc(sizeof(struct example_state), GFP_KERNEL);
      if (state == NULL) {
              dev_err(dev, "failed to create our state\n");
              return -ENOMEM;
      }

      state->client = client;
      i2c_set_clientdata(client, state);

      /* rest of the initialisation goes here. */

      dev_info(dev, "example client created\n");

      return 0;
}

static int example_remove(struct i2c_client *client)
{
      struct example_state *state = i2c_get_clientdata(client);

      kfree(state);
      return 0;
}

static struct i2c_device_id example_idtable[] = {
      { "example", 0 },
      { }
};

MODULE_DEVICE_TABLE(i2c, example_idtable);
```

(continues on next page)

(continued from previous page)

```
static struct i2c_driver example_driver = {
      .driver         = {
              .owner          = THIS_MODULE,
              .name           = "example",
              .pm             = &example_pm_ops,
      },
      .id_table       = example_idtable,
      .probe          = example_probe,
      .remove         = example_remove,
};
```

6.2 I2C device driver binding control from user-space in old kernels

Note: Note: this section is only relevant if you are handling some old code found in kernel 2.6. If you work with more recent kernels, you can safely skip this section.

Up to kernel 2.6.32, many I2C drivers used helper macros provided by <linux/i2c.h> which created standard module parameters to let the user control how the driver would probe I2C buses and attach to devices. These parameters were known as `probe` (to let the driver probe for an extra address), `force` (to forcibly attach the driver to a given device) and `ignore` (to prevent a driver from probing a given address).

With the conversion of the I2C subsystem to the standard device driver binding model, it became clear that these per-module parameters were no longer needed, and that a centralized implementation was possible. The new, sysfs-based interface is described in How to instantiate I2C devices, section “Method 4: Instantiate from user-space”.

Below is a mapping from the old module parameters to the new interface.

6.2.1 Attaching a driver to an I2C device

Old method (module parameters):

```
# modprobe <driver> probe=1,0x2d
# modprobe <driver> force=1,0x2d
# modprobe <driver> force_<device>=1,0x2d
```

New method (sysfs interface):

```
# echo <device> 0x2d > /sys/bus/i2c/devices/i2c-1/new_device
```

6.2.2 Preventing a driver from attaching to an I2C device

Old method (module parameters):

```
# modprobe <driver> ignore=1,0x2f
```

New method (sysfs interface):

```
# echo dummy 0x2f > /sys/bus/i2c/devices/i2c-1/new_device
# modprobe <driver>
```

Of course, it is important to instantiate the `dummy` device before loading the driver. The dummy device will be handled by i2c-core itself, preventing other drivers from binding to it later on. If there is a real device at the problematic address, and you want another driver to bind to it, then simply pass the name of the device in question instead of `dummy`.