Linux I2C Driver

Content

- Linux I2C Terminology
- Linux I2C Subsystem
- Linux I2C Drivers
  - I2C Adapter/Master/Bus Drivers
  - I2C Device Drivers
  - I2C Client/Slave Interface
- Instantiating I2C Devices
- User space tools

Kernel Space

LINUX I2C SUBSYSTEM

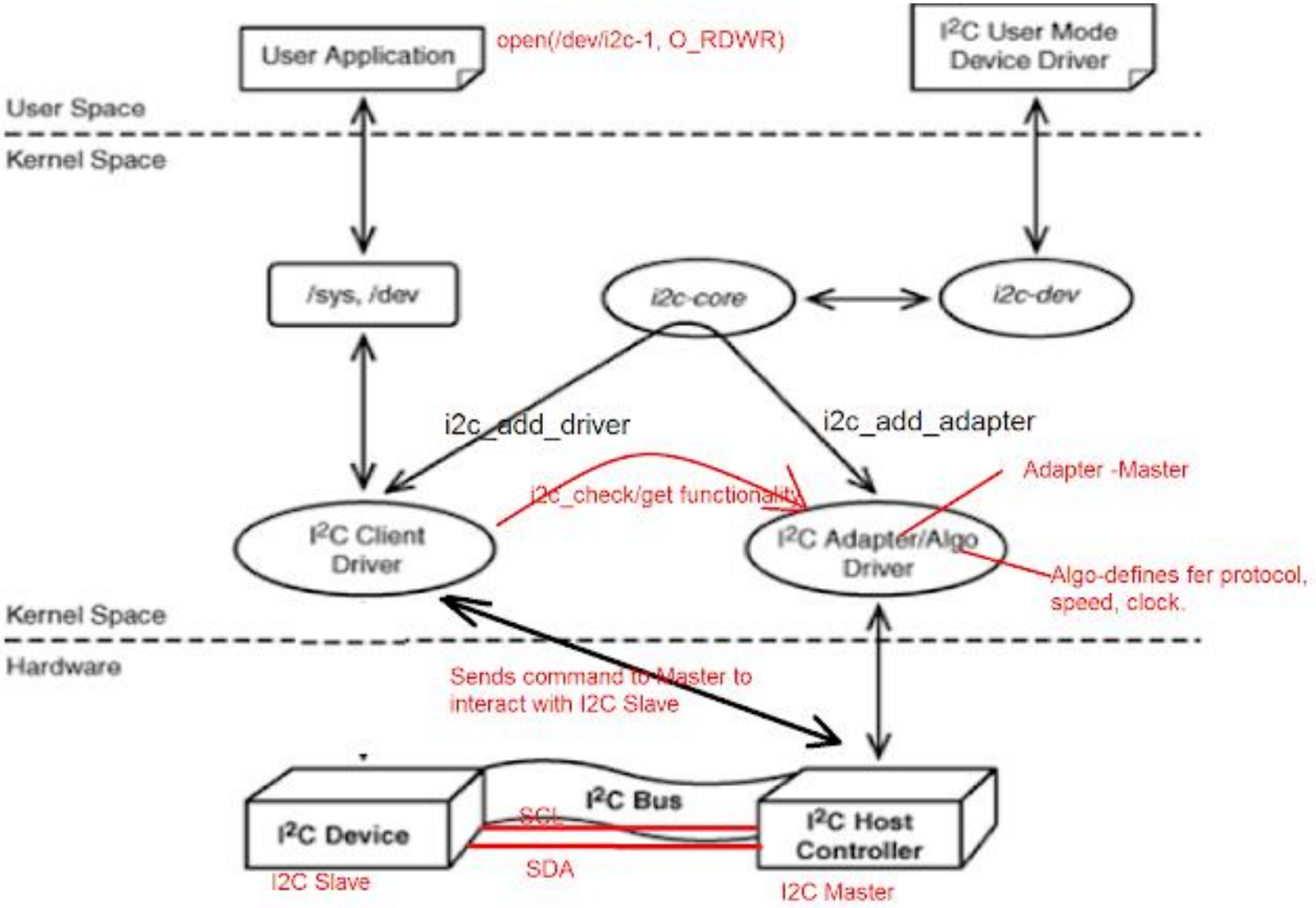
Linux I2C Terminology

- **I2C Master/Adapter/Bus Driver**
  - 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.
- **I2C Algorithm**
  - 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.

Linux I2C Terminology

- **I2C Slave/Client Driver**
  - 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 to the feature they provide,
  - for example
  - “driver/rtc/” for RTC drivers,
  - “drivers/input/touchscreen” for Touchscreen drivers,
  - “drivers/iio/adc/” for ADC expanders,
  - “drivers/media/gpio/” for GPIO expanders,
  - “drivers/media/i2c/” for video-related chips.

Linux I2C SubSystem



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.

I2C/SMBUS FUNCTIONALITY

I2C/SMBUS FUNCTIONALITY

i2c_check_functionality() : Verifies whether a particular function is supported by the host adapter.

i2c_get_functionality() : Obtains a mask containing all function

I2C/SMBUS FUNCTIONALITY

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

Kernel Space

I2C ADAPTER/MASTERS/BUS DRIVER

I2C Bus Driver

- Define and allocate a private data struct (contains **struct i2c_adapter**)

- Fill algorithm struct
  - **.master_xfer()** – function to perform transfer
  - **.functionality()** – function to retrieve bus functionality.

- Fill adaptor struct
  - **i2c_set_adapdata()**
  - **.algo** – pointer to algorithm struct
  - **.algo_data** – pointer the private data struct

- Add adapter
  - **i2c_add_adapter()**

Kernel API for I2C Adapter (master)

```
/*
 * i2c_adapter is the structure used to identify a physical i2c bus along  with the access
algorithms necessary to access it.
 */
struct i2c_adapter {
        struct module *owner;
        unsigned int class;              /* classes to allow probing for */
        const struct i2c_algorithm *algo; /* the algorithm to access the bus */
        void *algo_data;

        /* data fields that are valid for all devices   */
        const struct i2c_lock_operations *lock_ops;
        struct rt_mutex bus_lock;
        struct rt_mutex mux_lock;

        int timeout;                    /* in jiffies */
        struct device dev;              /* the adapter device */

        int nr;
        char name[48];
        struct completion dev_released;

        struct i2c_bus_recovery_info *bus_recovery_info;
};
```

```
/* struct i2c_algorithm - represent I2C transfer method */

struct i2c_algorithm {
/* master_xfer should return the number of messages successfully
        processed, or a negative value on error */
 int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg
*msgs, int num);

  int (*master_xfer_atomic)(struct i2c_adapter *adap, struct
i2c_msg *msgs, int num);

   int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr, unsigned
short flags, char read_write, u8 command, int size, union
i2c_smbus_data *data);

  int (*smbus_xfer_atomic)(struct i2c_adapter *adap, u16 addr,
unsigned short flags, char read_write, u8 command, int size, union
i2c_smbus_data *data);

        /* To determine what the adapter supports */
        u32 (*functionality) (struct i2c_adapter *);

};
```

Kernel Space

I2C DEVICE DRIVER

I2C Device Drivers

- Device → Driver
  Client
- A Driver driver (yes, this sounds ridiculous, sorry) contains the general code to access some type of device.
- Each detected device gets its own data in the Client structure. Usually, Driver and Client are more closely integrated than Algorithm and Adapter.

Kernel API for I2C Driver

struct i2c_driver: Represents of i2c_driver

```
static struct i2c_driver foo_driver = {
        .driver = {
                .name   = "foobar",
                .of_match_table = of_match_ptr(foo_dt_ids)
        },
        .id_table       = foo_idtable,
        .probe          = foo_probe,
        .remove         = foo_remove,
};
```

Kernel API for I2C Driver

struct i2c_client: Identifies a chip connected to an I2C bus

```
struct i2c_client {
        unsigned short flags;          /* div., see below              */
        unsigned short addr;           /* chip address - NOTE: 7bit    */
                                       /* addresses are stored in the  */
                                       /* _LOWER_ 7 bits               */
        char name[I2C_NAME_SIZE];
        struct i2c_adapter *adapter;   /* the adapter we sit on        */
        struct device dev;             /* the device structure         */
        int init_irq;                  /* irq set at initialization    */
        int irq;                       /* irq issued by device         */
        struct list_head detected;
};
```

Kernel API for I2C Driver

i2c_add_driver() *:* Registers driver entry points with the I2C core.

i2c_del_driver() : Removes a driver from the I2C core.

Initializing the I2C 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.

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

I2C SLAVE INTERFACE

HOW TO INSTANTIATE I2C DEVICES

How to instantiate I2C devices

Method 1: Declare the I2C devices statically [Declare the I2C devices via device tree]

```
&i2c1 {
     pinctrl-names = "default";
      //pinctrl-0 = <&i2c0_pins>;
      pinctrl-0 = <&i2c1_pins>;

      status = "okay";
      clock-frequency = <100000>;
      rtc1: ds1307@68 {
          compatible = "dallas,ds1307";
             reg = <0x68>;
             #address-cells = <1>;
             #size-cells = <1>;
       };
};
```

For syntax: Documentation/devicetree/bindings/i2c/i2c-omap.txt

How to instantiate I2C devices

i2c_device_id / of_device_id

```
static struct i2c_device_id foo_idtable[] = {
        { "foo", 0 },
        { }
};

MODULE_DEVICE_TABLE(i2c, foo_idtable);

static const struct of_device_id foo_dt_ids[] = {
    { .compatible = "foo,bar", .data = (void *) 0xDEADBEEF },
    { }
};
MODULE_DEVICE_TABLE(of, foo_dt_ids);
```

How to instantiate I2C devices

Declare the I2C devices in board files:

```
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));
        (...)
}
```

How to instantiate I2C devices

Method 2: Instantiate from user-space:

Sysfs interface is made of 2 attribute files which are created in every I2C bus directory: ``new_device`` and ``delete_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.

How to instantiate I2C devices

Method 2: Instantiate from user-space:

Example:

Add New Slave Devices:

$ echo eeprom 0x50 > /sys/bus/i2c/devices/i2c-1/new_device

$ echo ds1307 0x68 > /sys/bus/i2c/devices/i2c-1/new_device

Delete Slave Devices:

$ echo 0x50 > /sys/bus/i2c/devices/i2c-1/delete_device

$ echo 0x68 > /sys/bus/i2c/devices/i2c-1/delete_device

Kernel Space

I2C CORE

KERNEL MASTERS

Kernel SMBUS I2C API for I2C Core

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);

Kernel Plain I2C API for I2C Core

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.

User Space

LINUX I2C DEVICE DRIVERS IN USER SPACE

User Space Tools

Simple character device driver (i2c-dev)

- Device nodes at /dev/i2c-x
- Slave address set by I2C_SLAVE ioctl.
- Simple access using read() / write()
- i2c_smbus_{read,write}_{byte,word}_data()

i2ctools

- i2cdetect
- i2cget
- I2cset

User Space Tools

From the Linux user space, you can access the I2C bus from the **/dev/i2c-*** device files.

```
debian@beaglebone:~$ ls -l /dev/i2c-*
crw-rw---- 1 root i2c 89, 0 Oct 7 16:40 /dev/i2c-0
crw-rw---- 1 root i2c 89, 1 Oct 7 16:40 /dev/i2c-1
crw-rw---- 1 root i2c 89, 2 Oct 7 16:40 /dev/i2c-2
```

User Space Tools

List I2C devices on the bus

```
debian@beaglebone:~$ i2cdetect -y -r 2
0 1 2 3 4 5 6 7 8 9 a b c d e f
00: -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- 1c -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

User Space Tools

Dump the register contents of MMA8453

```
debian@beaglebone:~$ i2cdump -y -r 0x00-0x31 2 0x1c
No size specified (using byte-data access)
0 1 2 3 4 5 6 7 8 9 a b c d e f 0123456789abcdef
00: ff fe 00 01 80 41 80 00 00 00 00 01 00 3a 00 00 .?.??A?....?.:..
10: 00 80 00 44 84 00 00 00 00 00 00 00 00 00 00 00 .?.D?...........
20: 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 ..........?.....
30: 00 00
```

User Space Tools

Read and write single registers of the MMA8453

```
debian@beaglebone:~$ i2cget -y 2 0x1c 0x0d
0x3a
debian@beaglebone:~$ i2cget -y 2 0x1c 0x2a
0x00
debian@beaglebone:~$ i2cset -y 2 0x1c 0x2a 0x01
debian@beaglebone:~$ i2cget -y 2 0x1c 0x2a
0x01
```