A comprehensive study and analysis for development of an I2C Linux Device Driver

KARNA D. GAJAB
Electronics and Communication, Gujarat Technological University, L.D College of Engineering Campus, Navarangpura, Ahmedabad, INDIA - 380015
karna0490@gmail.com

ABSTRACT
Technology in recent times has witnessed an exponential evolution of compact and speedy devices to cater higher end applications. In order to continue the trend, it is utmost necessary to regulate and monitor the onboard communication and performance of the system/device. In context to this fact, I2C is one of the primary solutions as a reliable protocol for serial communication across short circuit board distances connecting Main Processing Units with their sub-processing units. Additionally, device drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time dependent hardware interface. In relation to this fact, LINUX kernel based OS is an excellent solution. In Linux environments, programmers can build device drivers either as parts of the kernel or separately as loadable modules.

This paper comprehensively illustrates the development flow of an I2C Linux Device Driver in modern high speed SoC based systems, wherein high performance through efficient on-board communication is a necessity.

Keywords: I2C, Linux, Device Driver, Kernel, module, board support package, LDD

[1] INTRODUCTION
The need and demand of device drivers in the current technological front has elevated drastically. There is a constant requirement for developing suitable device drivers and integrating them to a suitable target system for successful functional utility of a SoC (system on chip). The most preferred platform for development of the drivers is the Linux environment and the Linux drivers are popularly coined with the term LDD (Linux Device Driver) As multiple IC’s on a single SoC have the constant requirement to exchange signal levels and communicate, a dedicated on board serial communication protocol is the need of the hour. Efficient protocols like I2C (Inter – Integrated Circuit) are preferred to cater to the cause.

This paper discusses the I2C protocol and its use, a basic understanding and usage of Linux Device Drivers and methodology on developing an I2C Linux Device Driver for a target system.

[2] THE I2C PROTOCOL
Inter-Integrated Circuit, generically referred to as “two-wire interface” is a multi-masterserial single-ended computer bus invented by Philips that is used to attach low-speed peripherals to a motherboard, embedded system, or cell phone or other electronics. The I2C bus is a standard communication protocol that the ICs in today’s electronic applications employ for performing communication functions between intelligent control devices (e.g. microcontrollers), general-purpose circuits (e.g. LCD drivers, remote I/O ports, memories) and application-oriented circuits (e.g. digital tuning and signal processing circuits for radio and video systems).

2.1 The Applicability
In the world of communication protocols today, FC is often considered as a ‘little’ communication protocol compared to Ethernet, USB, SATA, PCI-Express and others, that present throughput in the x100 megabit per second range if not gigabit per second. Today, the I2C bus is used in many other application fields than just audio and video equipment. The bus is generally accepted in the industry as a de-facto standard. The I2C bus has been adopted by several leading chip manufactures like Xicor, ST Microelectronics, Infineon Technologies, Intel, Texas Instruments, Maxim, Atmel, Analog Devices and others.

2.1.1 I2C Protocol features
The I2C bus physically consists of 2 active wires and a ground connection. The active wires, called SDA and SCL, are both bi-directional. SDA is the Serial DAta line, and SCL is the Serial Clock line. The I2C bus is a multi-master bus. This means that more than one IC capable of initiating a data transfer can be connected to it. The I2C protocol specification states that the IC that initiates a data transfer on the bus is considered the Bus Master. Consequently, at that time, all the other ICs are regarded to be Bus Slaves. There are four potential modes of operation for a given bus device, although most devices only use a single role and its two modes:

- o master transmit — master node is sending data to a slave
- o master receive — master node is receiving data from a slave
- o slave transmit — slave node is sending data to the master
- o slave receive — slave node is receiving data from the master
Table 1: I2C Protocol – Byte scheme/Data frame

<table>
<thead>
<tr>
<th>Field No.</th>
<th>Length</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>1-bit</td>
<td>It is the start bit.</td>
</tr>
<tr>
<td>2nd</td>
<td>7-bits</td>
<td>It is called the address field. It defines the slave address to or from which the data is being sent or accessed.</td>
</tr>
<tr>
<td>3rd</td>
<td>1-control bit</td>
<td>It defines whether a read or write cycle is in progress.</td>
</tr>
<tr>
<td>4th</td>
<td>1-control bit</td>
<td>This bit defines whether the present data is a n acknowledgement (from the slave).</td>
</tr>
<tr>
<td>5th</td>
<td>8-bits</td>
<td>It is used for the IC device data bits.</td>
</tr>
<tr>
<td>6th</td>
<td>1-bit</td>
<td>It is a negative acknowledgement bit (NACK) from the master. If active, then acknowledgement after a transfer is not needed from the slave, else acknowledgement is expected from the slave.</td>
</tr>
<tr>
<td>7th</td>
<td>1-bit</td>
<td>It is the stop bit.</td>
</tr>
</tbody>
</table>

2.1.2 Modes of Operation

The I2C protocol generally works in the following two modes of operation:

- **Enhanced I2C (Fast Mode):** In the FAST mode, the physical bus parameters are not altered. The protocol, bus levels, capacitive load etc. remain unchanged. However, the data rate has been increased to 400 Kbit/s and a constraint has been set on the level of noise that can be present in the system. To accomplish this task, a number of changes have been made to the I2C bus timing.

- **High Speed I2C (HS Mode):** High-speed mode (Hs-mode) devices offer a quantum leap in I2C-bus transfer speeds. Hs-mode devices can transfer information at bit rates of up to 3.4 Mbit/s, yet they remain fully downward compatible with Fast- or Standard-mode (F/S-mode) devices for bi-directional communication in a mixed-speed bus system. With the exception that arbitration and clock synchronization is not performed during the Hs-mode transfer, the same serial bus protocol and data format is maintained as with the F/S-mode system. Depending on the application, new devices may have a Fast or Hs-mode I2C-bus interface, although Hs-mode devices are preferred as they can be designed-in to a greater number of applications.

2.1.3 Start and Stop conditions

Prior to any transaction on the bus, a START condition needs to be issued on the bus. The start condition acts as a signal to all connected IC’s that something is about to be transmitted on the bus. As a result, all connected chips will listen to the bus.

After a message has been completed, a STOP condition is sent. This is the signal for all devices on the bus that the bus is available again (idle). If a chip was accessed and has received data during the last transaction, it will now process this information (if not already processed during the reception of the message).

- **SDA** / **Start** The chip issuing the Start condition first pulls the SDA (data) line low, and next pulls the SCL (clock) line low.
- **SDA** / **Stop** The Bus Master first releases the SCL and then the SDA line. A few notes about start and stop conditions:

2.1.4 Transmitting a byte to a slave device

Once the start condition has been sent, a byte can be transmitted by the MASTER to the SLAVE. This first byte after a start condition will identify the slave on the bus (address) and will select the mode of operation. The meaning of all following bytes depends on the slave.

**Note:** Even in 10-bit extended addressing mode, Bit 0 of the first byte after the Start condition determines the slave access mode (‘1’ = read / ‘0’ = write).
### 2.1.5 Receiving a byte from a slave device

Once the slave has been addressed and the slave has acknowledged this, a byte can be received from the slave if the R/W bit in the address was set to READ (set to ‘1’). The protocol syntax is the same as in transmitting a byte to a slave, except that now the master is not allowed to touch the SDA line. Prior to sending the 8 clock pulses needed to clock in a byte on the SCL line, the master releases the SDA line. The slave will now take control of this line. The line will then go high if it wants to transmit a ‘1’ or, if the slave wants to send a ‘0’, remain low.

![Fig 3 - Byte scheme during transmission](www.esaacademy.com)

All the master has to do is generate a rising edge on the SCL line (2), read the level on SDA (3) and generate a falling edge on the SCL line (4). The slave will not change the data during the time that SCL is high. (Otherwise a Start or Stop condition might inadvertently be generated.) During (1) and (5), the slave may change the state of the SDA line. In total, this sequence has to be performed 8 times to complete the data byte. Bytes are always transmitted MSB first.

![Fig 4 - Reception byte scheme representation](www.esaacademy.com)

### 3. DEVICE DRIVERS – VITALITY AND TYPES

A device driver is a program that a computer’s operating system uses to control a particular hardware device that is attached to a computer. Devices can range from the display and hard disk drive, to sound cards and video cards. Most operating systems contain device drivers that are built-in, but they can also be installed when a new device is added to the system.

#### 3.1 Classification of Device Drivers

There are several kinds of device drivers, each handling a different kind of I/O. Block device drivers manage devices with physically addressable storage media, such as disks. All other devices are considered character devices.

- **Block Device Drivers**:
- **Character Device**:
- **Bye Stream I/O**:

![Fig 6 - User-Hardware interaction via Device Driver](www.esaacademy.com)
4. REQUIRED SOFTWARE SUITE TO BUILD THE I2C DEVICE DRIVER

Following is a list of all the necessary components required as part of the software suite to build the driver:

- I2C Driver Code (preferably in C)
- Embedded Linux as the OS
- A stable Linux Kernel version e.g. 2.6.35
- Suitable Boot Loader for the hardware:
  - Primary Boot Loader – e.g. XLoader
  - Secondary Boot Loader – e.g. UBoot
- GNU G++ compiler
- Code Sourcery/Tool Chain
- Operating File System (Minimal if the driver has to be ideal)
- Necessary packages and dependencies if the file system has to be customized
- A serial terminal for configuration of the hardware for which the driver has to be ported

Of all the above listed components, except the I2C Driver code, rest all can be encompassed and termed singly as “BSP – Board Support Package”.

5. DEVELOPING THE DRIVER AS A MODULE FOR THE LINUX KERNEL

While it may, in many cases, be desirable to provide static-kernel code for a driver, during development recompiling and rebooting to test the code every time is time-consuming. It is therefore highly desirable to develop the code as a module.

The following is a short example code illustrating how a device driver can be accessed by the user:

```c
// Access a device driver from userland
int fd;
unsigned char ucBuff [ PREDEFINED_SIZE ];
/* open the device */
fd = open ("/dev/my_device", O_RDWR);
/* read back some data from it */
read (fd, ucBuff, PREDEFINED_SIZE);
```

After the C program, which is the device driver source code has been developed; it has to be compiled via the GCC Compiler in Linux, which is a software compiler package of the GNU Tool Chain.

Using the terminal, type the following code:

```
$ gcc -c module-name.c
```

This will produce our ELF binary module-name.o.

This should be then linked into the kernel by typing:

```
$ insmod module-name.o
```

Now the module should be loaded into the kernel. We can verify the inserted modules by listing them with the command:

```
$ lsmod
```

For unloading any inserted module, we can type the following:

```
$ rmmod module-name
```

In the subsequent section, a sample code has been taken for development into a module and the use of Linux Kernel module creation process has been significantly illustrated in the following stages:

- **Stage 1**: Writing a module_name.c program file
- **Stage 2**: Writing a corresponding makefile for the module_name.c file
- **Stage 3**: Creating a module_name.o (object) file for the module insertion into the kernel
- **Stage 4**: Inserting the module into the kernel space (using the .ko extension)
- **Stage 5**: Removing the module from the kernel space (using the .ko extension)

6. DRIVER ARCHITECTURE

The device driver development steps can be listed as follows:

- An interface to the control and status registers.
- Variables to track the current state of the physical and logical devices
- Major and minor device number, device name
- A routine to initialize the hardware to known state
- An API for users of the device driver -- Read, write, and seek
- Interrupts service routines
  - For every module in Linux, three files are always included viz.
    - `#include <linux/module.h>`
    - `#include <linux/init.h>`
    - `#include <linux/moduleparam.h>`
  - `<linux/module.h>` stores the definitions of symbols and functions.
  - `<linux/init.h>` is used for initialization and function clean-up.
  - `<linux/moduleparam.h>` enables parsing of parameters to modules at load time.

### Fig 7 – Driver Architecture

To specify which license applies to code: The licensing part within a driver in Linux is always written at the end. Following is the licensing content:

- `MODULE_LICENSE (“GPL”);`
- `MODULE_AUTHOR();`  
- `MODULE_DESCRIPTION();`  
- `MODULE_VERSION();`  
- `MODULE_ALIAS();`  
- `MODULE_DEVICE_YABLE();`

*The above marked are compulsory to be written at the end of the Linux Device Driver code.

#### 6.1 DRIVER INITIALIZATION

The initialization of any driver can be a “wrapper” or can be a complete driver code inside. For an example: `Module_init (init_func);` can be treated as a wrapper for

```c
static int __init init_func (void)
{
    // code description area
}
```

The initialization function is always declared as “static” so that it is not visible outside the specific file. The call should compulsorily be from within the file. The module loader will drop the `init ()` function after the initialization process and will release the allocated memory. All the current processes are defined in the `<asm/current.h>` file, yielding a pointer to a structure `task_struct` inside `<linux/sched.h>`. `<current.h>` also refers to the processes running currently, invoked via system call like `open()`, `read()`, `write()`, `close()` etc. To get information of current processes, it can be done via the following command:

```bash
Printk (“KERNINFO’ process is %s
“(pid %i 

To get information of current processes, it can be done via the following command:

```bash
Printk (“KERNINFO’ process is %s
“(pid %i 
```

Any make file for the Driver code will be of the following type:

```bash
// A generic make file
KERNELDIR: = /lib/modules/ $(shell uname –r)/build
Obj -m : = <driver_name.o>
PWD : = $(shell, pwd)
default:
   $ (MAKE) -CS (KERNELDIR)
   M = $(PWD) modules
clean: rm –vf<driver_name.o> <driver_name.ko>
```

#### 7. THE I2C STACK

In general, the Linux 2.6 kernel contains an I2C driver stack that is split up into three layers:

1. **Chip driver:** a device-dependent part which interacts between user space and the core module.
2. **Core module**: a device-independent part containing an implementation of the I2C protocol.

3. **Bus driver**: a device-dependent part which interacts between the core module and the actual hardware. The core module is part of the Linux kernel, as are a number of chip drivers and bus drivers.

![High level layer structure of the I2C Linux Driver Stack](fig8)

*(Courtesy: Behavioural analysis of an I2C Linux Driver: DraganBosnacki, AadMathijsen, Yaroslav S. Usenko)*

### 7.1.1 The driver functionality and the I2C Subsystem

The I2C core is a code base consisting of routines and data structures available to host adapter drivers and client drivers. Common code in the core makes the driver developer's job easier. The core also provides a level of indirection that renders client drivers independent of the host adapter, allowing them to work unchanged even if the client device is used on a board that has a different I2C host adapter. This philosophy of a core layer and its attendant benefits is also relevant for many other device driver classes in the kernel, such as PCMCIA, PCI, and USB. In addition to the core, the kernel I2C infrastructure consists of the following:

- Device drivers for I2C host adapters. They fall in the realm of bus drivers and usually consist of an *adapter* driver and an *algorithm* driver. The former uses the latter to talk to the I2C bus.
- Device drivers for I2C client devices,
- I2c-dev, which allows the implementation of user mode I2C client drivers.

### 8. FINAL I2C SUBSYSTEM FLOW

![The I2C Subsystem flow](fig10)
9. APPLICATIONS

Because of the diversity of modern hardware and operating systems, drivers operate in many different environments. Drivers may interface with:

- printers
- video adapters
- network cards
- sound cards
- local buses of various sorts — in particular, for bus mastering on modern systems
- low-bandwidth I/O buses of various sorts (for pointing devices such as mice, keyboards, USB, etc.)
- computer storage devices such as hard disk, CD-ROM and floppy disk buses (ATA, SATA, SCSI)
- implementing support for different file systems
- image scanners
- digital cameras

CONCLUSIONS

This paper discusses about the I2C protocol, Linux Device Drivers and finally gives a comprehensive overview on how to proceed towards the development of an I2C Linux Device Driver for a particular target platform. It exemplifies on how driver codes can be made into a insertion specific module within the Linux Kernel and how porting of the driver code can be made on the destination hardware. As a conclusion to the approach, a final flow for the I2C subsystem is depicted that maps all the concepts pertaining to the discussions and relates to how a protocol can be ported as a successful functional device driver.

ACKNOWLEDGEMENTS

I would sincerely like to thank Gujarat Technological University for coming up with an initiative like IDP, under which I completed research on this project and which enabled me to expand my horizon and eventually write this paper. Also special mention to all my college faculties who clarified my doubts during the research phase of the project, making my concepts really clear to successfully execute this project.

REFERENCES