HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

GRADUATION THESIS

Linux device driver developing methods and applications with UART, SPI, I2C protocols

TRAN DUY ANH
anh.td203872@sis.hust.edu.vn

VO BA THONG
thong.vb203887@sis.hust.edu.vn

Major in Embedded Systems and IoT

Instructor:	Assoc. Prof. Hoang Manh Thang	________________ Instructor's signature
Department:	Electronics	

HANOI, 6/2024

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

GRADUATION THESIS

Linux device driver developing methods and applications with UART, SPI, I2C protocols

TRAN DUY ANH

anh.td203872@sis.hust.edu.vn

VO BA THONG

thong.vb203887@sis.hust.edu.vn

Major in Embedded Systems and IoT

Instructor:	Assoc. Prof. Hoang Manh Thang	________________ Instructor's signature
Dissertation Committee:		
Department:	Electronics	

HANOI, 6/2024

PREFACE

In the world of embedded systems, device drivers are crucial for communication between the operating system and hardware. With the advent of new technology, there are more and more sophisticated devices created for high-end applications, which requires a thorough knowledge about the device driver development. With this knowledge, one can modify devices to serve a specific need or to meet the requirements of a particular system. Despite their importance, there is no unifying method for developing these drivers, particularly for UART, I2C, and SPI devices. This thesis is motivated by the need for a standardized approach to driver development.

By providing essential knowledge about the Linux Kernel and Linux Device Drivers, this work aims to create a general outline that can be applied across different devices. Our goal is to bridge the gap between theory and practice, offering a valuable resource for developers in the embedded systems field.

This thesis begins with an exploration of core concepts such as Kernel Modules, Character Device Drivers, Platform Device Drivers, and the Device Tree. These foundational elements establish a framework that can be utilized to create various device drivers, particularly for UART, I2C, and SPI devices. The later chapter presents practical implementations for these devices, illustrating the flexibility and efficiency of this approach.

ACKNOWLEDGEMENT

We would like to express our deepest gratitude to our supervisor, Assoc. Prof. Hoang Manh Thang, for the invaluable guidance and support throughout this thesis. Your expertise and insightful feedbacks have been instrumental in shaping our research. We are also profoundly thankful to our mentor, Mr. Vu Van Hung for his encouragement and the knowledge he imparted. Our heartfelt appreciation goes to our families for their unwavering supports and understanding, and to our friends for their constant encouragement and motivation. Your collective contributions and faith in us have been crucial in the completion of this work.

Students

Tran Duy Anh Vo Ba Thong

DECLARTION OF AUTHORSHIP

We are: Tran Duy Anh, student ID of 20203872, and Vo Ba Thong, student ID of 20203887, both are students of Cohort 65 School of Electrical and Electronic Engineering, majoring in Embedded systems and IoT.

We hereby declare that this thesis was carried out by ourselves under the guidance and supervision of Assoc. Prof. Hoang Manh Thang. All of the data and results presented in it are completely truthful, accurately reflecting the results of the real-world simulation. All cited information complies with intellectual property regulations; the referenced documents are clearly listed. We take full responsibility for the content written in this thesis.

Hanoi, 27th of June, 2024

Authors

Tran Duy Anh **Vo Ba Thong**

TABLES OF CONTENTS

LIST OF FIGURES

LISTS OF TABLES

LIST OF ABBREVIATIONS

BPS	Bits Per Second
CD-ROM	Compact Disc Read-Only Memory
CPU	Central Processing Unit
CS	Chip Select
DMA	Direct Memory Access
DT	Device Tree
DTB	Device Tree Blob
DTS	Device Tree Source
EEPROM	Electrically Erasable Programmable Read-Only Memory
ext3	Third Extended File System
FAT	File Allocation Table
GNU	GNU’s Not Unix
I2C	Inter-Integrated Circuit
I2S	Integrated Inter-IC Sound Bus
LCD	Liquid Crystal Display
LDM	Linux Device Model
MOSI	Master Out Slave In
MISO	Master In Slave Out
OOP	Object-Oriented Programming
PCI	Peripheral Component Interconnect standard
SATA	Serial Advanced Technology Attachment
SCLK	Serial Clock
SS	Slave Select
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver - Transmitter
USB	Universal Serial Bus

TÓM TẮT ĐỒ ÁN

Đề tài đồ án là "Xây dựng phương pháp phát triển Linux device driver và ứng dụng vào các thiết bị sử dụng giao thức UART, SPI, I2C". Cấu trúc bao gồm ba phần:

- **Phần 1: Giới thiệu tổng quan**: Phần này trình bày về tổng quan về Linux device drivers, mục tiêu và phạm vi của đề tài, và phương pháp tiếp cận đề tài.
- **Phần 2: Nội dung chính**: Phần này bao gồm hai chương:
  - Chương 1: Chương này tập trung vào:
    - Trình bày các kiến thức nền tảng về Linux kernel và Linux device driver;
    - Trình bày cấu trúc của character device drivers cùng với các thành phần cơ bản của chúng và một vài công cụ cần thiết;
    - Trình bày về khái niệm Device Tree và platform device drivers;
    - Và ghép tất cả các nội dung trên và đưa ra khung phát triển character device drivers.
  - Chương 2: Chương này trình bày hướng triển khai của các thiết bị cụ thể bao gồm:
    - Module cảm biến vân tay điện dung FPC1020A sử dụng giao thức UART, triển khai trên Raspberry Pi 3 Model B;
    - AT24C256 EEPROM sử dụng giao thức I2C, và LCD NOKIA5110 sử dụng giao thức SPI, triển khai trên Beagle Bone Black.
  - Chương 3: Chương này trình bày kết quả triển khai các thiết bị trên.
- **Phần 3: Kết luận**: Phần này sẽ tổng kết đề tài, đánh giá mức độ hoàn thiện của drivers đã triển khai và đề xuất định hướng phát triển trong tương lai.

Phân chia công việc	**Sinh viên**
Lý thuyết cơ bản Linux device driver	Tran Duy Anh
Character device drivers	Tran Duy Anh
Platform device drivers	Vo Ba Thong
Linux Device Model	Vo Ba Thong
Triển khai trên thiết bị UART	Tran Duy Anh
Triển khai trên thiết bị SPI và I2C	Vo Ba Thong

ABSTRACT

The thesis topic is “Linux device driver developing methods and applications with UART, SPI, I2C protocols”. The structure consists of three parts:

- **Part 1: Introduction**: This section presents the overview of Linux device drivers, the aim and range of the topic, and the method of approach of the thesis.
- **Part 2: Main content**: This section consists of two chapters:
    - Chapter 1: This chapter focuses on:
        - Presenting fundamental knowledge about the Linux kernel and Linux device drivers;
        - Presenting the structure of character device drivers with their basic components and some tools needed;
        - Presenting the concept of Device Tree and platform device drivers;
        - And putting everything all together to create an outline for developing character device drivers.
    - Chapter 2: This chapter presents implementation outlines and results of specific devices which are:
        - FPC1020A Capacitive Fingerprint Sensor module using UART protocol, on a Raspberry Pi 3 Model B;
        - AT24C256 EEPROM using I2C protocol, and LCD NOKIA5110 using SPI protocol on a Beagle Bone Black.
    - Chapter 3: This chapter presents implementation results of the above devices
- **Part 3: Conclusion**: This section is to summarize the topic, evaluate the completion level of the deployed drivers and propose future development directions.

Work assignment	**Student**
Linux device driver basis	Tran Duy Anh
Character device drivers	Tran Duy Anh
Platform device drivers	Vo Ba Thong
Linux Device Model	Vo Ba Thong
Deployment on UART device	Tran Duy Anh
Deployment on SPI and I2C devices	Vo Ba Thong

INTRODUCTION

Overview

The Linux kernel is a complex, portable, and widely used piece of software, which is popular in many embedded systems in more than half of devices throughout the world. In which, device drivers play a crucial role in how well a Linux system performs. With the rising trend of innovative hardware products, there is a growing need for someone to work on ensuring that all these new devices are compatible and functional with the Linux operating system. However, working with Linux is like walking into a deep dark wood without a path, since Linux is an evolving operating system so it is kind of large and overwhelming for those who are new to it. That is why we need to create an easy-to-understand approach to developing Linux device driver.

Aim and range of topic

The topic of the thesis is "Linux device driver developing methods and applications with UART, SPI, I2C protocols". The aim of the topic is to:

- Learn how to expose a device's functionalities to many different user space applications;
- Learn how the Linux kernel works and how to adapt to its working;
- Create a general outline of developing a Linux device driver which is easy to understand and approach;
- Show how that outline is applied on real-life character devices to make them work with the Linux kernel;
- And learn all basic the skills that a developer needs in developing device drivers.

Method of approach

The method of approach is to start with a simple character device driver to understand what are the core parts of a general device driver; next, based on that driver skeleton, we add up more driver's "LEGO" pieces that add more functionality to our driver like adding protocol communication, user space and kernel space communication, and so on; then, we assemble all of those components and create a general outline of writing device drivers that can be applied on many different devices; and finally, we apply that outline on our chosen devices.

The building and testing of the device drivers can be performed on a Linux based computer with any character device available. For this topic, we've chosen to develop drivers for: an FPC1020A Fingerprint sensor module, a NOKIA5110 LCD screen, and AT24C256 EEPROM; which is performed on a Raspberry Pi 3 single-board computer.

CHAPTER 1. LINUX CHARACTER DEVICE DRIVER BASIS

This chapter’s main focus is on: giving some background concepts in developing with Linux kernel environment; giving an overview about the structure of a basic driver skeleton; providing developers with basic tools and macros needed to start developing Linux kernel modules, specifically character device modules; building a structure of character device driver from the basic skeleton mentioned above; demonstrating platform device drivers and why they are used; and exploring Linux Kernel Model to understand what components lie beneath the surface and how things work together.

1.1 Linux device driver basis

Linux is a family of open-source Unix-like operating systems based on the Linux kernel, created by Linux Torvalds in 1991. Linux is typically packaged as a Linux distribution, which includes the kernel and supporting software and libraries.

“Linux offers many advantages over other operating systems:

- It is free of charge
- Well-documented with a large community
- It is portable across different platforms
- It provides access to the source code
- There is lots of free, open source software [1]”

A kernel is a central part of an operating system that generally has complete control over everything in the system. The Linux kernel is a large and complex body of code so kernel programmers need an entry point where they can approach the code without being overwhelmed by its complexity. Device drivers could be a good starting point.

Device drivers hold a unique and critical position within the Linux kernel. They can be described as “black boxes” that control specific hardware pieces to enable them to respond to a well-established internal programming interface. In doing so, they completely obscure the details of how those devices actually function. It can be defined that a device driver is a piece of software that exposes the functionalities of the hardware to other programs. From an operating system point of view, drivers can be either in the kernel space (running in privileged mode) or in the user space (with lower privileges). Our aim is to build a kernel space driver. User interactions are performed by a collection of standardized function calls that are independent to the specific driver. These calls are then mapped to device-specific operations that operate in the underlying layer of real hardware. This programing interface is such that drivers can be built separately from the rest of the kernel and plugged in when needed during runtime. In other words, this is the Linux drivers' modularity, which has allowed many of the drivers available and easy to write [2].

1.1.1 Userspace and kernel space

Before deep-diving into the world of kernel, it is more recommended to first have an overview of the fundamental architecture of Linux, user space and kernel space.

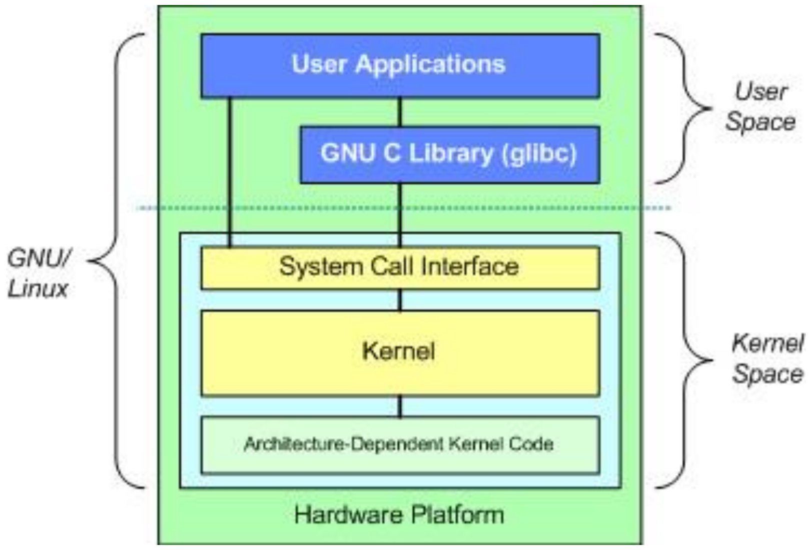


Figure 1.1 Fundamental architecture of GNU/Linux operating system

Figure 1.1 introduces the separation between *user space* and *kernel space*, and highlights the fact that there is a bridge between them, which is the *system call* interface.

In reality, the architecture may not as simple as what is shown in Figure 2.1, but it is meant to be like that since it is easier for us to visualizing the mechanism, in this case, how the *system calls* are handled from the *user space* to the *kernel space*. As they are a bit abstract since they are all about memory and access rights, each term will be covered and described in the following sections.

1.1.1.1. User space

User space is a set of addresses where normal programs or applications are restricted to run. Examples of these programs are: text editors, web browsers, and so on. User space may be considered as a sandbox or even a jail, as a user program is not allowed to mess with memory addresses or any other resources owned by another program.

User space applications cannot directly access the system's hardware resources unless they make *system calls* to the kernel to request access to these resources. Examples of these are *read, write, open, close,* and so on. User space codes run with a low priority.

1.1.1.2. Kernel space

Kernel space (sometimes called *kernel memory*) is a set of addresses where the kernel is hosted and where it runs. It is a memory range that is owned by the kernel and protected by access flags so it prevents any user applications from messing with the kernel without being unknowable.

On the contrary to user applications, the kernel can access the whole system memory, since it runs with the highest priority on the Linux system.

1.1.1.3. Separation of user space and kernel space

The separation of user space and kernel space is at the base of design principle in operating systems. The purposes of this separation, in practice, are:

- Security: User space applications won't accidentally or maliciously corrupt the kernel or other system resources.
- Stability: Operating system stays safe and stable even when there are potential failures in user space applications.
- Performance: It can improve the performance of the operating system by allowing the kernel to run in a protected environment.

1.1.2 Splitting the kernel

In a Unix system, several concurrent processes attend to different tasks. Each process asks for system resources, and come the *kernel* handling all such requests. "Although the distinction between the different kernel tasks is not always clearly marked [2]", to make it easier to understand, we could split it into the following parts:

- *Process management:* The kernel is generally in charge of creating, destroying, scheduling processes and handling their connections to the outside world (input and output). In addition, the scheduler, which is part of the process management, controls how processes share the CPU. More generally, the kernel's process management activity implements the abstraction of several processes on top of one or a few CPUs [2].
- *Memory management:* The computer's memory is a major resource that the absence of a managing policy would result in a critical hindrance in the system performance. The kernel builds up a virtual addressing space for every process on top of the limited available resources. The different parts of the

kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple *malloc/free* pair to much more complex functionalities [2].

- *Filesystems:* “Unix is heavily based on the concept of filesystem; almost everything in Unix can be treated as a file [2].” The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. Furthermore, Linux is capable of working with a variety of filesystem types, which in turn implies that there are diverse methods for structuring data on the same physical storage medium. For example, disks may be formatted with ext3 filesystem, FAT filesystem or several others.
- *Device control:* Nearly every system-level operation maps back to an underlying physical platform. Excluding the processor, memory, and a small number of other components, each and every device control operations are carried out by code that is specifically “tailored” to the target device. This “specialized” piece of code is known as a *device driver*. For the kernel to properly manage a system's peripherals - from the hard drive to the keyboard and beyond - it must have an embedded device driver for each one. This device driver functionality represents a core aspect of the kernel's responsibilities that we will be focusing on in this topic.
- *Networking:* Networking must be managed by the operating system, since most network operations are not specific to a process; incoming packets are asynchronous events. Data packets must first be gathered, classified, and directed before a process can handle them. The system is responsible for delivering data packets across application and network interfaces, as well as resolving routing and addressing concerns.

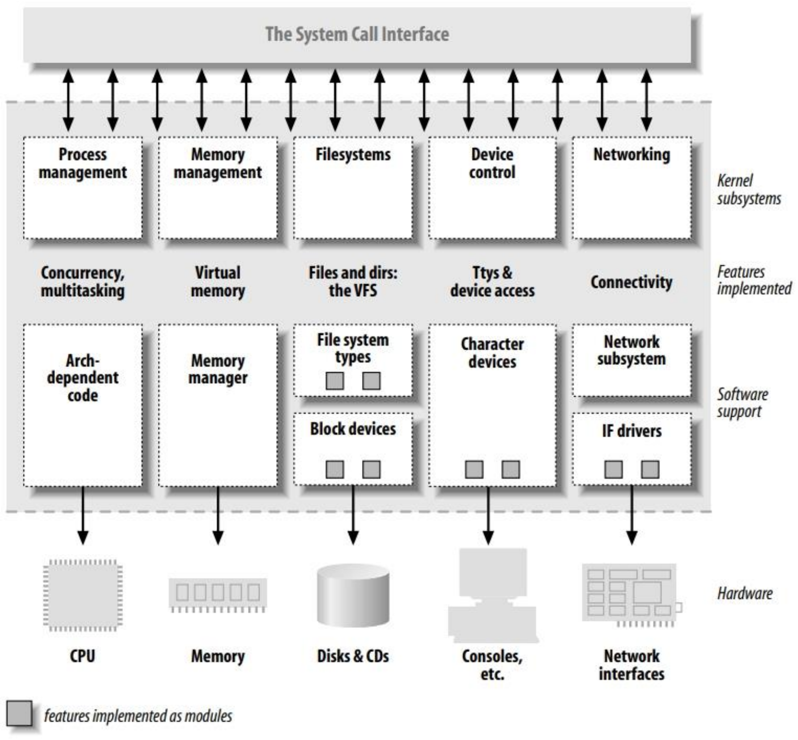


Figure 1.2 A split view of the kernel [2]

Figure 1.2 shows a split view of the kernel, in which different tasks are divided into different subsystems that we have mentioned above. Each subsystem implements different features.

1.1.3 Concept of modules

Linux allows programmers to have the ability to extend the set of features offered by the kernel during its runtime. This means that it is possible to add or remove functionalities to the kernel while the system is at work (running).

"Each piece of code that can be added to the kernel at runtime is called a *module* [2]." In other words, what a module is to the Linux kernel is like what a plugin (add-on) is to user software [3]. The system can be dynamically extended with functionalities without having to restart. In addition, the Linux kernel offers support for quite a few different types of (or classes) of modules, including, but not limited to, device drivers [2].

Figure 2.2 covers some of the most important classes but it is far from complete because more and more functionalities in Linux are being modularized [2].

From this point forward, the way of calling drivers and modules are interchangeable, because a driver is a module that runs in the kernel that do the talk with some hardware device and that is the aim of this thesis anyway.

1.1.4 Classes of devices and modules

The way Linux look at devices can be classified into three fundamental device types: *character modules*, *block modules* and *network modules*. As mentioned above, more and more functionalities are being modularized, this would result that the division of modules into different types, or classes, is not always a fixed one. A programmer can choose to implement as many device driver as they want in a single chunk of module. Nevertheless, it is more recommended to create different modules for each new functionality or each device being implemented, because decomposition is a key element for a system to be scalable and extendable.

For the classes of module that were mentioned above, it is also possible to divide them into a few classes of device as follows.

1.1.4.1. Character devices

A *character device,* also known as *char device,* is a type of device that can be accessed and interacted with as a continuous stream of bytes, similar to how one would work with a file. So, a character driver in charge of implementing that behavior. Typically, such drivers implement at least the *open*, *close*, *read*, and *write* system calls. Character devices can be accessed through the filesystem nodes in the */dev* directory. The only relevant difference between a character device and a regular file is that we can always move back and forth in a regular file, whereas most character devices are just channels, which we can only access sequentially [2].

Examples of character devices are: keyboard, mouse, serial ports, sound card, joystick, etc.

For the topic of this thesis, the main focus is on developing drivers for this type of devices.

1.1.4.2. Block devices

A *filesystem* is controls how data is stored and retrieved in Linux operating system.

A *block device* is a device that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or larger but to the power of two) bytes in length [2]. As a result, block and character devices differ only in the way data is managed and internally stored.

Examples of block devices are: hard disk driver, CD-ROM, USB, etc.

Like character devices, block devices are also accessible by the filesystem nodes in the */dev* directory.

1.1.4.3. Network interfaces

Any network-based transaction must occur through an interface capable of exchanging data with other systems. Typically, a network interface is a physical hardware component, but it can also sometimes be a virtual, software-based interface, such as the loopback interface. “A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted [2].”

A network interface isn’t easily mapped to a node in the filesystem since it is not a stream-oriented device. The Unix way to provide access to it is by assigning a unique name but that name doesn’t have a corresponding entry in the filesystem.

1.1.5 Kernel modules versus Applications

One more thing to consider before starting to work with drivers is to understand how it is different between a kernel module and a user space application. The differences varies from execution, to structure and data transfer [4].

Table 1.1 Differences between kernel modules and applications

	Kernel modules	**Applications**
Address space	Run in kernel space	Run in user space
Execution privilege	High privilege	Low privilege
Execution order	Do not execute sequentially; register themselves in order to serve future request	Execute sequentially; perform a single task from beginning to end
Event-driven programming	Every module	Not all applications
Exit function	Carefully undo initialized resources	Can be lazy in releasing resources or avoid clean up

Loadable	Can be loaded and unloaded to the kernel easily	Cannot be linked into the kernel
Libraries and header files	Can only call those are exported by the kernel	Vary
Error handling	A segmentation fault kills the current process, if not the whole system	A segmentation fault is harmless and can be traced using debugger

The Table 2.1 indicates a few key differences which pointed out that kernel modules and user applications, though may sometimes perform the same task, are meant to play different roles and how those roles are performed, especially their privileges.

1.1.6 Driver skeleton

As we have figured out what is a kernel module, what is its purposes and how it is different from the user space application, it is time to take a look at a basic skeleton of a Linux kernel module. First, we begin with a “hello world” example since this is a way of showing the simplest possible program.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE(“GPL”);

static int __init helloworld_init(void)
{
    printk(KERN_ALERT “Hello World!\n”);
    return 0;
}

static void __exit helloworld_exit(void)
{
    printk(KERN_ALERT “Goodbye, cruel world!\n”);
}

module_init(helloworld_init);
module_exit(helloworld_exit);
```

Figure 1.3 A simple “hello world” module

Figure 1.3 shows an example of a simple module, in which defines two functions `helloworld_init` and `helloworld_exit`. The function `helloworld_init` runs when the module is loaded into the kernel and the function *helloworld_exit* runs when that module is removed. These functions implement a *printk* function that prints out info to notice the kernel, which comes in handy especially when debugging the module. There are three special macros used in this example: `module_init`, `module_exit`, and `MODULE_LICENSE`. `module_init` and `module_exit` macros are used to indicate the roles of functions we mentioned above, while `MODULE_LICENSE` is used to tell the kernel about the license of this module, which is a free one. One more thing can be noticed is that this module uses header files under the Linux kernel source, which is a key difference that we mentioned prior to this part.

From this example, we could visualize it into a structural skeleton as shown below.

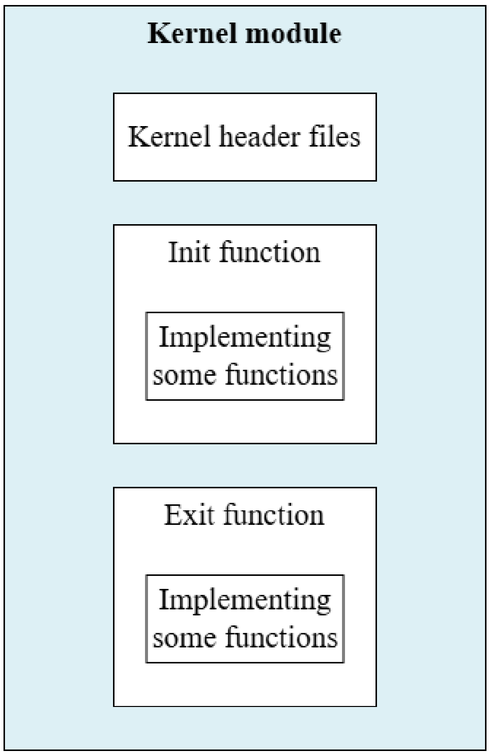


Figure 1.4 Basic skeleton of a kernel module

Figure 1.4 is a way of demonstrating a simple skeleton of a Linux kernel module, which makes it easier to understand the structure of the module rather than watching it in plain codes. A module can be imagined like as a “frame” of a “house”. As progressing further and needing more functionalities, more parts are added to the “frame” like “doors” or “windows”. For each new functionality, we will add it as a block in the module. This is our main method of approach from now on, specifically in building up a character device driver.

1.1.7 Module entry and exit points

Every kernel module must have an entry point and an exit point. This is corresponding to the *insmod* and *rmmod*. We will take a look on these functions when in the *Loading and unloading modules* part later.

In a user space program written in C or C++, the entry point is the `main()` function, and the exit point is when that same function returns.

With kernel modules, things are different. It is required to inform the kernel which functions should be executed as an entry or exit points. The actual functions `helloworld_init` and `helloworld_exit` that are shown in Figure 1.3 are these points and we can give them any name we want. The only thing that is mandatory is that we have to give them as parameters to the `module_init()` and `module_exit()` macros.

```
static int __int initialization_function(void)
{
    /* Initialization code here */
}
module_init(initialization_function);
```

Figure 1.5 Definition of initialization function

```
static void __exit exit_function(void)
{
    /* Cleanup code here */
}
module_init(exit_function);
```

Figure 1.6 Definition of exit function

The behaviors of the *init* and *exit* functions depend on what resources should be initialize upon entry and removed upon exit, as showcased in Figure 1.5 and 1.6. In the case of Figure 1.3, we just inform the kernel when is loaded and unloaded. More complex drivers will implement more behaviors and functionalities.

1.1.8 Message printing

This part has nothing to do with the kernel module skeleton or any functionality implementation, but it is worth noticing because knowing how to handle errors and print them out make it easier for kernel developers to debug and speed up the kernel development process.

It can be seen in Figure 1.3 that there is a function `printk()`.`printk()` is, to the kernel, what `printf()` is to the user space. Messages written by `printk()` are printed to the kernel log buffer, which is a ring buffer exported to user space through */dev/kmsg*; and they can be displayed using the *dmesg* command. Depending on how important the message we need to print out, we can choose between eight log levels defined in *include/linux/kern_levels.h*.

When we develop our device driver, it is recommended to use `printk()` anytime that we can, since it is a good tool to trace and debug.

1.1.9 Compiling modules

Once we have done with adding up components to make our basic kernel module work, it is time to know how to build the module from kernel source to make it executable.

The build process for kernel modules differs significantly from user space applications. The kernel is a large, standalone program with detailed and explicit requirements on how its pieces are put together. The build process also differs from how things were done with previous versions of the kernel; the new build system is simpler to use and produces more correct results, but it looks very different from what came before. So before compiling anything, just make sure to take into account the kernel's version. The files found in the *Documentation/kbuild* directory in the kernel source are required reading for anybody wanting to understand all that is really going on beneath the surface.

There are some prerequisites that must get out of the way before we can build the kernel modules. The first is to ensure that we have sufficiently current versions of the compiler, module utilities, and other necessary tools. Trying to build a kernel and its module with non-proper tools could lead to hard-to-solve problems. The next thing is to make sure the kernel tree come in handy. We cannot build loadable modules without this tree on our filesystem.

Once all everything above is set up, we are now ready to create a makefile for our module. To make the case specific, we will create a makefile for the helloworld.c module that we mentioned in Figure 1.3.

```
obj-m += helloworld.o

KDIR = /lib/modules/$(shell uname -r)/build

all: module

module:

    make -C $(KDIR) M=$(shell pwd) modules

clean:

    make -C $(KDIR) M=$(shell pwd) clean
```

Figure 1.7 Makefile script for the helloworld.c module

Normally when we build our kernel module, we can type the *make* command it directly in the terminal, but it is tiresome after a while. A solution to this is by creating a makefile script, as Figure 1.7 has shown in example.

A makefile is a special file used to execute a set of actions, among which the most important is the compilation of programs [3].

In almost every kernel makefile, we will see at least one instance of an `obj-<X>` variable, where `<X>` should be either `y`, `m`, or `n` [3]. In Figure 1.7, the option is `obj-m`. This tells the kernel build utility that there is one object in the current directory named *helloworld.o* that will be built from *helloworld.c*. The option `<X>` decides how and whether *helloworld.o* module will be built or linked:

- If `<X>` is set to *m*, *helloworld.o* module will be built as a module.
- If `<X>` is set to *y*, *helloworld.o* module will be built as a part of the kernel (built-in).
- If `<X>` is set to *n*, *helloworld.o* module will not be built.

The line `KDIR = /lib/modules/$(shell uname -r)/build` is the location of the prebuilt kernel source. This is used to build our loadable module. The line `all: module` instructs the *make* utility to invoke the `module` target. In other words, *make all* or just simply *make* commands will just be translated into *make modules*. In our targets, we define the rules to be executed for each target. In the case of Figure 1.7, there are 2 rules, one in the *module* target is to build our modules, while the other one in the *clean* target is to clean up our module object. The resulting module of the build process is named *helloworld.ko,* which is named after the target object file we declared.

We must remember that the makefile script can vary and be adjusted to work best with the build process of our specific modules.

1.1.10 Loading and unloading modules

The usual command to remove or unload a module from the kernel is *rmmod*. This command is preferred to unload a module that is loaded with the *insmod* command. Both of these commands should be given the module name (the insmod command requires full name with the file extension in this case is *.ko)*. For example, in the case of Figure 1.3 and 1.7, we will use *insmod helloworld.ko* for loading and *rmmod helloworld* for unloading the module.

On the other hand, there is *modprobe*, which is another method of loading and unloading modules. *modprobe* will parse the modules' *.dep* files to load dependencies first, prior to loading the given modules. In order to use this command, we must have administrative privilege to manage the kernel modules and add our modules to the */lib/modules/<kernel_version>* directory. For example, in the case of Figure 1.3 and 1.7, we will use *modprobe helloworld* for loading and *modprobe -r helloworld* for unloading.

Another thing that could be really helpful sometimes is to use *lsmod* to check whether a module is loaded or not.

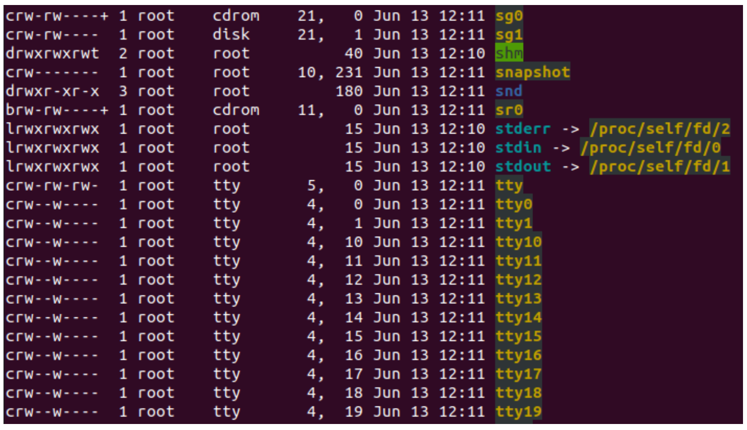
1.2 Character device drivers

1.2.1 Major and minor numbers

1.2.1.1. The concept of major and minor

There is a basic concept in Linux that says *everything is a file*. “Character devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the */dev* directory [5].” These special files for character drivers can be seen when we use the command *ls -l*, and are identified by the letter “c” in the first column of the output. Block devices also appear in the */dev* directory as well, but identified by the letter “b”. The possible values of file types can be:

- c: for character device files;
- b: for block device files;
- l: for symbolic link;
- d: for directories;
- s: for sockets;
- p: for named pipes.



```
crw-rw----+ 1 root    cdrom    21,   0 Jun 13 12:11 sg0
crw-rw----  1 root    disk     21,   1 Jun 13 12:11 sg1
drwxrwxrwt  2 root    root          40 Jun 13 12:10 shm
crw-------  1 root    root     10, 231 Jun 13 12:11 snapshot
drwxr-xr-x  3 root    root         180 Jun 13 12:11 snd
brw-rw----+ 1 root    cdrom    11,   0 Jun 13 12:11 sr0
lrwxrwxrwx  1 root    root          15 Jun 13 12:10 stderr -> /proc/self/fd/2
lrwxrwxrwx  1 root    root          15 Jun 13 12:10 stdin -> /proc/self/fd/0
lrwxrwxrwx  1 root    root          15 Jun 13 12:10 stdout -> /proc/self/fd/1
crw-rw-rw-  1 root    tty       5,   0 Jun 13 12:11 tty
crw--w----  1 root    tty       4,   0 Jun 13 12:11 tty0
crw--w----  1 root    tty       4,   1 Jun 13 12:11 tty1
crw--w----  1 root    tty       4,  10 Jun 13 12:11 tty10
crw--w----  1 root    tty       4,  11 Jun 13 12:11 tty11
crw--w----  1 root    tty       4,  12 Jun 13 12:11 tty12
crw--w----  1 root    tty       4,  13 Jun 13 12:11 tty13
crw--w----  1 root    tty       4,  14 Jun 13 12:11 tty14
crw--w----  1 root    tty       4,  15 Jun 13 12:11 tty15
crw--w----  1 root    tty       4,  16 Jun 13 12:11 tty16
crw--w----  1 root    tty       4,  17 Jun 13 12:11 tty17
crw--w----  1 root    tty       4,  18 Jun 13 12:11 tty18
crw--w----  1 root    tty       4,  19 Jun 13 12:11 tty19
```

Figure 1.8 Example of *ls -l* command

As we can see in Figure 1.8, the first column identifies many types of many devices, like the *tty** are character devices, or the *sr0* is a block device. In addition, the fifth and sixth columns right before the date column are for representing *major* and *minor* numbers of a device file. So what are these numbers and what are they for?

We can understand that this is just simply a mechanism of kernel to determine and control exactly which device file is associated with which device driver. The *major* number identifies which driver associated with the device; and the *minor* number identifies which device is referred to. To make things less confusing, many devices can share the same driver, which means sharing the same *major* number, so the *minor* is there to tell which one of them we are referring to.

The kernel holds the major and minor numbers of a device in a `dev_t` type variable. This is a 32-bit unsigned long, in which, 12 bits are set aside for the major, and 20 remaining bits are for the minor, both are defined in */linux/types.h*. To obtain the major and minor, we can use a set of macros: `MAJOR(dev_t dev)` and `MINOR(dev_t dev)`. If, instead, we have the major and minor and need to turn them into a `dev_t`, we can use: `MKDEV(int major, int minor)`.

1.2.1.2. Allocating and freeing device numbers

One of the first things that we need to set up in a character device driver is to obtain one or more device numbers to identify our device file across the system. There are two ways to allocate these numbers: statically and dynamically.

1.2.1.2.1. Static allocation

If we want to set a particular major and minor number, we will do as follows.

First, we use the `MKDEV(major, minor)` macro to assign a variable with the major and minor numbers that we want.

```
dev_t dev = MKDEV(211, 0);
```

Figure 1.9 Example of assigning device number

Next, we will use a function called `register_chrdev_region()`, which is declared in */linux/fs.h*. The prototype of this function is as follows:

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

in which, we will pass the `dev_t` variable into this function as a parameter, along with `count` as the number of consecutive devices, and `name` as the associated device or driver. The return value of this function will be 0 if the allocation was successfully performed or will be a negative error code if the allocation failed. Example is as follows:

```
register_chrdev_region(dev, 1, "my_char_dev");
```

Figure 1.10 Example of registering device number statically

This method is only recommended if we know ahead of time that exactly which device numbers we want. However, in reality, we often do not know which major our device will use. That is why the dynamic method is more recommended.

1.2.1.2.2. Dynamic allocation

If we don't want fixed major and minor numbers, we will use the function `alloc_chrdev_region()`. Its prototype is as follows:

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
```

in which, `dev` is an output-only parameter that will hold the first number in our allocated range. `firstminor` should be the requested first minor number to use, usually 0. The `count` and `name` parameters are the same as the static method.

1.2.1.2.3. Freeing allocated numbers

Regardless of which method we used to allocate, we must free them whenever device numbers are no longer in use with the following function:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

usually, we place this function in our module's cleanup function, which can be module exit function that we mentioned earlier, or the probe remove function that we will discuss later.

1.2.2 *Some important data structures*

Now, we will take a look at an important data structure that normally exists in a standard character device driver – file operations structure.

We have reserved device numbers for our drivers, but we have not connected any of our drivers' operations to them yet. That is why the kernel uses the `struct file_operations` to expose a set of callbacks that will handle userspace system calls on a file [6]. For example, when a user perform a `read` system call, the callback function that corresponds to it will be called. Our job as a device driver developer here is to implement those callback functions and tie them to our device.

```
struct file_operations {
     struct module *owner;
     ssize_t (*read) (struct file *, char __user *, size_t,
loff_t *);
     ssize_t (*write) (struct file *, const char __user *,
size_t, loff_t *);
     int (*open) (struct inode *, struct file *);
     long (*unlocked_ioctl) (struct file *, unsigned int,
unsigned long);
     int (*release) (struct inode *, struct file *);
     [...]
};
```

Figure 1.11 struct file_operations

Figure 1.11 only lists some important methods of the structure that are mostly used in many drivers. The full description of every method is covered in */include/linux/fs.h*. Each of these callbacks are linked with a system call, and none of them is mandatory. When a user code calls a file-related system call on a given file, the kernel looks for the driver responsible for that file, locates its struct file_operations structure, and checks whether the method that matches the system call is defined or not. If yes, it simply runs. If not, it returns an error code that varies depending on the system call [6].

Inside these callback functions, especially in `write` and `read` callbacks, they normally implement some functions in order to copy a buffer from user space to kernel space, and backward, which are `copy_from_user()` and `copy_to_user()`. However, details of how we implement operations should be left to the programmers based on how they want their drivers to work.

1.2.3 *Character device allocation and registration*

Character devices are also represented in the kernel in the form of `struct cdev`, defined in */include/linux/cdev.h*. *cdev* is an element of the inode structure, and is set and used to indicate that the inode that we are referring to is a character device.

When writing a character device driver, our goal is to create and register an instance of that structure associated with `struct file_operations` in order to expose a set of operations (functions) that the user space can perform on the device [6]. To reach that goal, we will go through some steps as follows:

- Reserve a major and a minor (or a range of minors) with `alloc_chrdev_region()` function.
- Create a class for our device(s) with `class_create()`, to make our class visible in */sys/class*.
- Set up a struct file_operations to give to `cdev_init()`, and for each device that we need to create, we call `cdev_init()` and `cdev_add()` to register the device.
- Then, use `device_create()` for each device, with a proper name. It will result in our device being created in the */dev* directory.
- On clean up function, we must remember to use `cdev_del()`.

1.2.4 Memory allocation

For implementing more functionalities, like creating more structs for implement some specific devices, one might also want to use `kmalloc` and `kfree` for the allocation and freeing memory of those. Actually, Linux offers an even richer set of memory allocating primitives, but those are based on how `kmalloc` and `kfree` work, so there is no need to cover them all.

To those who have been familiar with the user space's `malloc`, `kmalloc` is the kernel version of it; it will allocate a physical memory region in the kernel. The same thing goes with `kfree`.

Details of how to implement these functions will be shown in the applicating section of this topic.

1.2.5 Sleeping and waitqueue techniques

Sleeping is a mechanism that can also be deployed in a kernel driver. Sometimes we might want our driver to have the ability to wait or sleep for some events, like changes in variables. There are many techniques with many utilities that can make it work, but we will just take a look at some simple sleeping mechanism to apply to our device drivers.

"The kernel scheduler manages a list of tasks to run, known as a run queue [7]." Whenever a process must wait for some kind of events, it moves out of our run queue and is suspended; and whenever the conditions are all met, that process will be waken up and it will continue its job. The way Linux provides this mechanism is by *wait queues*.

A *wait queue*, just what it sounds like, is a list a processes that all wait for a specific event [8]. In Linux, a wait queue is managed by a *wait queue head*. A wait queue head can be defined and initialized in two different methods:

- Static method: use the macro defined in */include/linux/wait.h*:
  `DECLARE_WAIT_QUEUE_HEAD(<wait_queue_name>)`
  in which, `wait_queue_name` is the name we give to the queue.
- Dynamic method: first declare a variable of `wait_queue_head_t` type and then use the `init_waitqueue_head()` function.

Once the wait queue is initialized, we can use several functions to put a process to sleep or wake it up, which are `wait_event_interruptible` and `wake_up_interruptible`. Both of these functions are defined as below:

```
int wait_event_interruptible(wait_queue_head_t q,
CONDITION);
```

```
void wake_up_interruptible(wait_queue_head_t *q);
```

in both of which, we will pass the queue that we declared as a parameter.

`wait_event_interruptible()` does not continuously poll, but simply just evaluates the condition when it is called. If the condition is not met, the process is put into a `TASK_INTERRUPTIBLE` state and removed from the run queue. The condition is rechecked each time we call `wake_up_interruptible()` in the wait queue. If the condition is true when `wake_up_interruptible()` runs, a process will be awakened with its state set to `TASK_RUNNING`. Processes are awakened in the order in which they are put to sleep. To awaken all process, we should use `wake_up_interruptible_all()`.

Instead of using sleeping and waitqueue techniques, in some cases like physical timing, a simple timer like delay would be more preferable. There are many delay functions like `ndelay`, `udelay`, `mdelay`, and so on, which come in to solve this problem. Since the use of these functions are based on what we want to implement, details of how they are used can be found in */include/linux/delay.h* and */include/linux/timer.h*.

1.2.6 ioctl system call

There are lots of system calls in Linux system, but only a few of them are linked with file operations. Sometimes, devices might want to implement something more specific than just `read` or `write`. For examples, we might want to have an operation that sends a command buffer to our device, or changes the baud rate of a serial port, which is clear that we have no particular system call by default. In cases like this, the solution is to use the *input/output control*, which is *ioctl* in short.

ioctl commands can be added to the existing file operations in our drivers, and they must be defined with a number that is unique across the whole system. This uniqueness will act as a way to prevent the conflict that occurs when the command is given a duplicated ioctl number. So before writing code for ioctl commands, we need to choose a "magic number" for our driver by checking */include/asm/ioctl.h* and *Documentation/ioctl-number.txt*. The *ioctl-number.txt* file lists the magic numbers that has already been used throughout the kernel, so by checking it we can choose our own magic number without overlapping.

After we have chosen a "safe" magic number, we can implement our ioctl commands with the following steps:

- Create ioctl command(s) in our driver with:
  ```
  #define <ioctl_name> __IOX(<magic_number>,
  <command_number>, <argument_type>);
  ```
  in which:
  - `X` can be `R` for read parameters, `W` for write parameters, or `WR` for both;
  - `magic_number` is the number that we have chosen;
  - `command_number` can be any number that we use to differentiate commands from one another;
  - And `argument_type` is the type of data of the argument we pass to the ioctl command.
- Write ioctl function in the driver with:
  ```
  long <ioctl_function_name> (struct file *filp,
  unsigned int cmd, unsigned long arg);
  ```
  in which:
  - `ioctl_function_name` can be any name given to the function we implement our ioctl system calls;
  - `filp` is a file pointer, which, in this case, is a file descriptor that represents our device file so user space applications can send data to it;
  - `cmd` is the command number that we assign to the commands;
  - `arg` is the argument we pass to the commands.

  Normally, we would use a standard C's `switch case` structure to switch between commands in case we have many of them. Detail of what is implemented inside those commands depends on the functions' purposes.

- Inform the kernel about the new ioctl by making the “`unlocked_ioctl`” pointer of the `struct file_operations` points to the ioctl function we declared:
  `.unlocked_ioctl = <ioctl_function_name>`.
- Next, define the same ioctl command(s) in step 1 in a userspace application. To make it easy, we can define them in a header file so we can include it in many applications, rather than just in a single source code.
- Last, call the ioctl command(s) when we want to use from a user application by using:
  `ioctl(<file_descriptor>, <ioctl_name>, <arg>);`
  in which:
  - `<file_descriptor>` is the open file that the ioctl command will be executed on, generally device files;
  - `<ioctl_name>` is the name of the ioctl command that we defined;
  - `<arg>` is the argument to pass to the *ioctl* command.

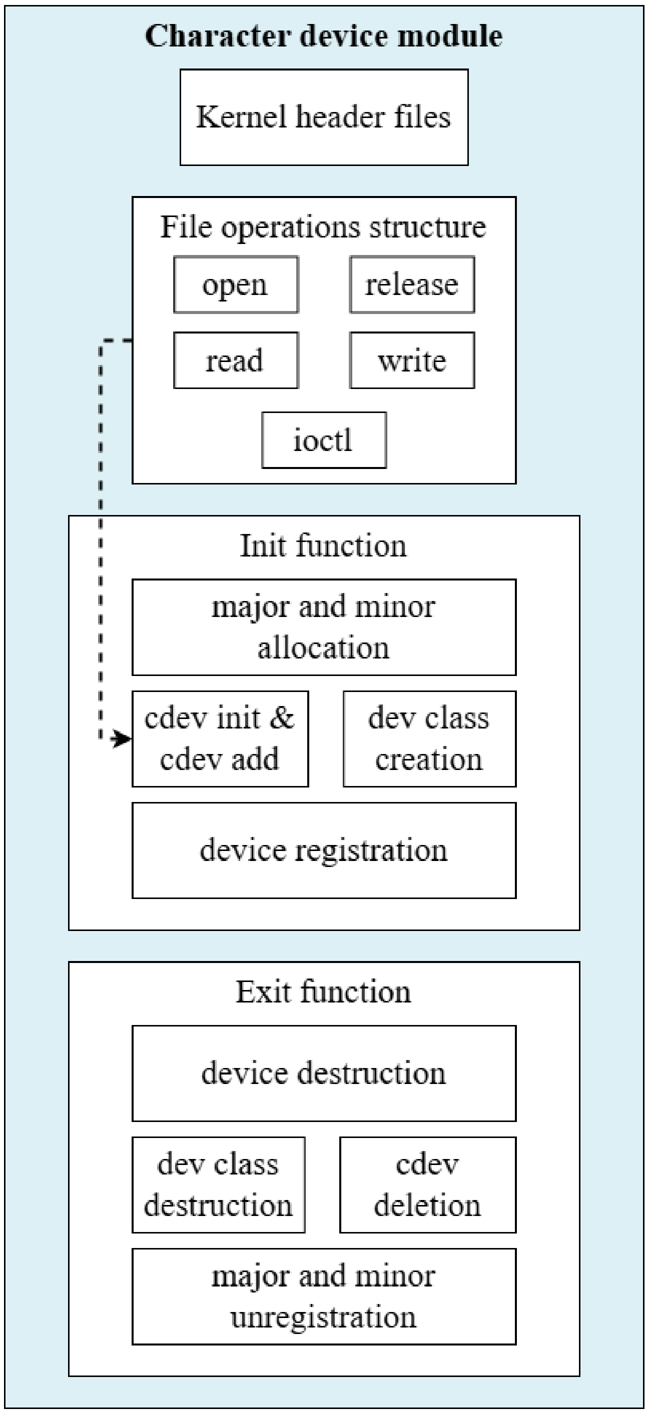


Figure 1.12 Character device module skeleton

Figure 1.12 is a representation of many components which are needed in a character device driver, and their initializations and removals. When comparing to Figure 1.4, it can be seen that these components are being added to the basic skeleton that was mentioned before.

1.3 Platform device drivers

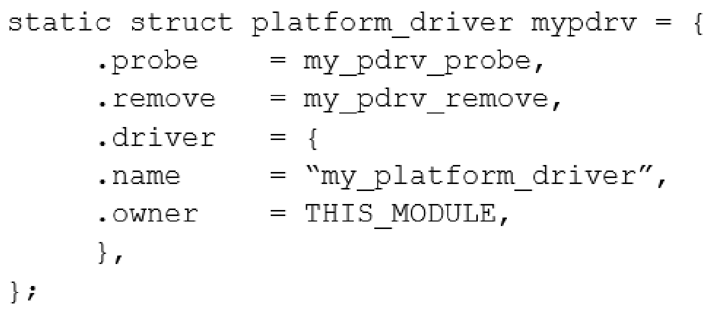
Up until now, the knowledge and tools needed for the interface between the user space and the kernel space has been provided but the interface between the kernel space and the device is still in mist. In order to make that interface enable, it requires some knowledge about the hardware, and it is hard because each devices has its own hardware configuration and there is no general device drivers for all devices up until Linux kernel version 2.6. In the next section we will introduce the platform device drivers, which served as a general structure to develop other drivers. Although the platform driver' original purpose was to create for platform devices, the devices which kernel needs to know about prior to being managed. The structure of this driver serves as a building block for develop other drivers.

1.3.1 Platform Device Drivers Skeleton

There are two main steps to create platform device driver:

- Register a platform driver that will manage
- Register platform device with their resources, in order to inform kernel about given device

The mandatory component is `struct platform_driver`, we need to register with the platform bus core with additional functions. An example of `struct platform_driver` is illustrated in the figure below.



```
static struct platform_driver mypdrv = {
     .probe    = my_pdrv_probe,
     .remove   = my_pdrv_remove,
     .driver   = {
     .name     = "my_platform_driver",
     .owner    = THIS_MODULE,
     },
};
```

Figure 1.13 Example of struct platform _driver

In the figure 1.13 above there are some important fields that needs attention to:

- `.probe`: Pointer to the callback `probe` function, which is called when there is a match between given device and driver;
- `.remove`: Pointer to the callback `remove` function, which is called to delete the driver when it is not needed by the device;
- `.driver`: Pointer to the `struct driver`, which describes the driver's information like: driver's name, owner, and some other fields.

To register platform driver with the kernel and platform devices we just simply need to call two function `platform_driver_register()` and `platform_driver_probe()` when the module is loaded. The former adds the driver into a list of drivers managed by the kernel, whenever a match occurs between the device and the driver, the driver's `probe()` function can be called. The `probe()` function is crucial because it responsible for initialize, allocate, configure the resources, data needed by the driver and device. The latter is called when if there is a platform device with the matching `compatible` field, which will be described more detail in the next Device Tree section. In case a match occurred, meaning that the device is present, if not the kernel ignores the driver.

The *probe* function and the *init* function deserve a proper attention. There is a little difference between them, which is demonstrated in the following table.

Table 1.2 Difference between the init function and the probe function

Init	***Probe***
Gets called whenever the module is loaded into the kernel	Gets called whenever there is a new device attached to the system and match with the appropriate driver
Called only once during the driver's lifetime	Can be called multiple times during the driver's lifetime
Manages global initialization of the driver module, sets up global resources, and registers the driver with the kernel	Manages initialization of specific device instances, allocates device-specific resources, configures hardware, and prepares the device for operations

```
static int my_pdrv_probe (struct platform_device *pdev) {
     pr_info("Device is probed successfully!\n");
     return 0;
}
static void my_pdrv_remove(struct platform_device *pdev) {
     pr_info("Good bye!\n");
}
static struct platform_driver mypdrv = {
     .probe    = my_pdrv_probe,
     .remove   = my_pdrv_remove,
     .driver   = {
                    .name     = KBUILD_MODNAME,
                    .owner    = THIS_MODULE,
     },
};
static int __init my_pdrv_init(void) {
     platform_driver_register(&mypdrv);
     return 0;
}
static void __exit my_pdrv_remove(void) {
     platform_driver_unregister(&mypdrv);
}
```

Figure 1.14 Example of platform device driver

The figure 1.14 above displays an example of platform device driver. If there is nothing more than register/unregister the driver with the platform bus core, we can use the `module_platform_driver` macro instead of `module_init` and `module_exit` macro, most drivers can use this scheme. In this macro there are still `init` and `exit` functions embedded in there, so it does not mean those functions are not needed anymore, just that it is more convenient for us to implement those functions. There are specific macros for each common bus, such as: `module_spi_driver` for SPI drivers, `module_i2c_driver` for I2C drivers, `module_pci_driver` for PCI drivers, `module_usb_driver` for USB drivers, `module_serdev_driver` for UART drivers,…

1.3.2 Device provisioning and Device Tree

1.3.2.1. Device Tree basics

When done dealing with registering driver, we proceed to the next step, which is registering platform device with their resources, in order to inform kernel about the given device. Normally, in the past, to configure the hardware setting and necessary resources such as I/O ports, memory regions, register offsets, IRQ line, DMA channels, bus,… embedded engineering need to create additional files which is bound to specific driver and define those parameters in there. This can be really tedious when it comes to update or modify some configurations. Any modification will require rebuilding the whole kernel. In order to make things simple and separate from the kernel source, the concept of the Device Tree was brought into the picture. The main goal of the Device Tree is to remove very specific and never-tested code from the kernel. The Device Tree is a hardware description file and has a format similar to a tree structure, where each node represents a device, and any node's property is equivalent to data or resource or configuration data of that device, in other words the Device Tree is the representation of the hardware architecture. This way, the only need is to recompile the Device Tree when some modifications is made [9].

The following is the first glance of Device Tree:

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second
string";
        a-byte-data-property = {0x01 0x23 0x34 0x56};
        child-node@0 {
            first-child-property;
            second-child-property = <1>;
            a-reference-to-something = <&node1>;
        };
    };
    label: nodename@reg {
        an-empty-property;
        a-cell-property = <1 2 3 4>;
        child-node@0 {
        };
    };
};
```

Figure 1.15 Example of Device Tree

1.3.2.1.1. Node label, name and address

Each node has a name in the format `<name>[@<address>]`, where `<name>` is a string up to 31 characters in length, and whether the node represents an addressable device or not, the `[@<address>]` is optional. The `label` is optional, labeling a node when there is a need to be referenced from a property of a another node, like a pointer in C programming language. A *pointer handle* (*phandle*) is a 32-bit value uniquely associated with a node so that the node can be referenced from a property in another node. Using `<&label>` to point to the node labeled `label`.

1.3.2.1.2. Data types in Device Tree

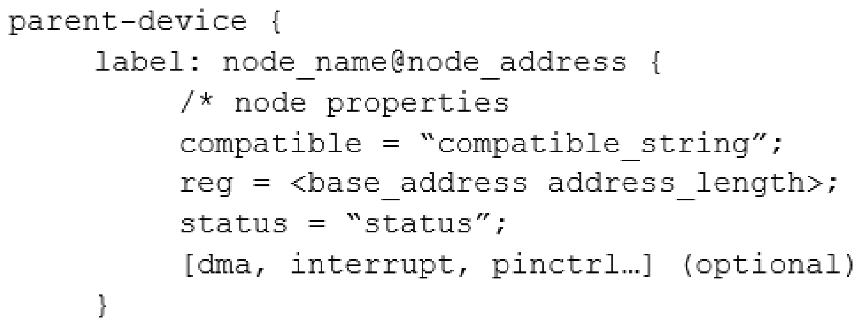
There are some most common data types used in Device Tree:

- Text strings enclosed with double quotes. A list of the strings is created by separating substring using commas;
- Cells which are 32-bit unsigned integers value represented with angle brackets;
- Byte data represented in HEX format and enclosed by square brackets and braces;
- Boolean data is no more than an empty property. Whether the property being there or nor decides the `TRUE` or `FALSE` value.

1.3.2.1.3. Forms of Device Tree and the device node

Device Tree has three forms: the textual form, which represents the sources, also known as Device Tree Source, the binary blob form with the *.dtb* extension, which represents the compiled Device Tree, also known as Device Tree Blob and the runtime representation of a Device Tree in */proc/device-tree*. Source files have the *.Device Tree* extension, there are also *.dtsi* text files, which represent SoC level definitions, whereas *.Device Tree* files represent board level definitions. Because the Device Tree is the representation of hardware architecture, *.dtsi* files are included in *.dts* files, not the reverse. The tool used to compile a device tree is called the *device tree compiler* (*dtc*). From the root kernel source, Compiling either a standalone specific Device Tree or all Device Tree for the specific architecture is up to the users' need.

Before we go further to the details, we want to provide the general device node, which served as an building block to register other device's nodes later on. The general device node is shown in the below figure.



```
parent-device {
    label: node_name@node_address {
        /* node properties
        compatible = "compatible_string";
        reg = <base_address address_length>;
        status = "status";
        [dma, interrupt, pinctrl…] (optional)
    }
```

Figure 1.16 General form of the device node

1.3.2.1.4. Representing and addressing devices

The `reg` property defines a list of memory region where we can access the device, which depends on the bus the device sits on. Each region is a couple of cells, in which the first cell is the base address of the region, and the second cell is the size of the region. The format of *reg* property is `reg = <base0 length0 [base1 length1] … >`. In addition, to fully define the `reg` property we need to know value of `#size-cell` and `#address-cells` which informs us how many 32-bit cells we must use to define an address, and how many 32-bit cells are used to represent size, respectively.

```
soc {
     #address-cells = <1>;
     #size-cells = <1>;
     compatible = "simple-bus";
     aips-bus@02000000 { /* AIPS1 */
          compatible = "fsl, aips-bus", "simple-bus";
          #address-cells = <1>;
          #size-cells = <1>;
          reg = <0x02000000 0x100000>;
          […];
     }
```

Figure 1.17 Example of addressing devices

The Figure 1.17 represents an example simple node device in the device tree, the `aips-bus` is one of the root devices, the devices which do not have parent devices. It needs one 32-bits cell to define an address, and one 32-bits cell to define size, and its memory region has the base address which is 0x02000000 and length of that region is 0x100000.

1.3.2.1.5. Handling resources

The microcontroller has lots of pins with different functions for each pin, in order to choose the mode a pin should work there comes the pin muxing mechanism. The subsystem takes responsibility for this is the *pin control (pinctrl).* The *pinctrl* subsystem provides:

- Pin muxing: Allows for reusing the same pin for different purposes;
- Pin configuration: Set up electronic properties of pins such as pull-up, pull-down, driver strength, debounce period, and so on.

The pin controller driver usually needs a set of two nested nodes to describe a group of pins configurations. The first node describes the function of the group, the second holds the pin's configuration. How pin groups are assigned in the Device Tree heavily depends on the platform, and the pin controller driver. A pin configuration node can be assigned to a device by means of two properties:

- `pinctrl-<ID>`: Allows to give the list of *pinctrl* configurations needed for a certain state of the device. It is a list of *phandles*, each of which points to a pin configuration node. These referenced pin configurations node must be child nodes of the pin controller that they configure;
- `pinctrl-name`: Allows for giving a name to each state in a list. List entry 0 defines the name for the integer state ID 0, list entry 1 for state ID 1, and so on. The state ID 0 is commonly given the name default [10].

The concept of named resources (*clock*, *irq*, *dma*, *reg*) is introduced to resolve the mismatches problem when the resources list is not ordered in a manner the driver expects. There are four commons properties to name the resources:

- `reg-names`: This is for a list of memory regions in the *reg* property;
- `clock-names`: This is for name clocks in the *clocks* property;
- `interrupt-names`: This is for each interrupt in the interrupts property;
- `dma-names`: This is for the *dma* property.

Let’s take a look in an example of a device node:

```
example_device {
     reg = <0x01000000 0x2000>, <0x3f400000 0x100>;
     reg-names = “region1”, “region2”;
     interrupts = <0 40 IRQ_TYPE_LEVEL_HIGH>, <0 35
IRQ_TYPE_LEVEL_HIGH>;
     interrupts-names = “abcd”, “efgh”;
     clocks = <&clks IMX6QDL_CLK_UART_IPG> <&clks
IMX6QDL_CLK_UART_SERIAL>;
     clocks-names = “ipg”, “per”;
     dmas = “&sdma 25 4 0>, <&sdma 26 4 0>;
     dma-names = “rx”, “tx”;

};
```

Figure 1.18 Example of named resources

With the device node in the above Figure 1.18 example, in the driver to extract named resources passing name of the resources as a parameter in some function like:

- `platform_get_resource_byname();`
- `platform_get_irq_byname().`

to get the right resource without bothering with the index.

One of the most important resources of devices is the interrupt. The interrupt is divided into two parts-the consumer side and the controller side and there are four properties used to describe interrupt in a Device Tree:

The controller is the device that exposes IRQ lines to the consumer. On the controller side, the device has the following properties:

- *interrupt-controller*: A Boolean property that should be defined in order to mark the device as being an interrupt controller;
- *interrupt-cells*: This is a property of interrupt controllers. It states how many cells are used to specify an interrupt for that interrupt controller.

The consumer is the device that generate the IRQ:

- *interrupt-parent*: For the device node that generates the interrupt, it is a property that contains a *phandle* pointer to the interrupt the controller node to which the devices is attached. If omitted, the device inherits that property from its parent node;

- *interrupts*: This is the interrupt specifier [11].

1.3.2.2. Platform device drivers and Device Tree

In order to match a device with its driver, the platform core uses the device tree's `compatible` property to match the device entry in `of_match_table`, which is a field of the `struct driver` substructure. Each device node has a `compatible` property, which is a string or a list of strings. A Device Tree match entry is described in the kernel as an instance of the `struct of_device_id` structure in which also has a `compatible` string to match the device node's compatible property in a Device Tree. Since the `of_match_table` is a pointer, passing an array of `struct of_device_id` to make given driver compatible with more than one device.

Devices are matched with their drivers by `compatible` string. According to the Linux Device Model (LDM), which will be described in more details in the next section, the bus element maintains a list of drivers and devices that are registered with it. Each time there is a new connection of driver or device to the bus, it starts the matching loop. If a match occurs, driver's `probe()` function will be executed immediately. In order to let the kernel know which devices are handled by which drivers, the macro, which enables the kernel to store devices' ID table in the device list maintained by the platform core, is *MODULE_DEVICE_TABLE*. So the kernel will traverse the Device Tree and the device list, if there is a compatible match, it will call the `probe()` function registered in the driver.

1.4 Linux Device Model

Prior to the advent of kernel version 2.6, the kernel does not have the mechanism to manage objects because they have little in common even though they sit on the same bus. With the advent of new bus types like PCI, a strong demand for a unifying model for the device driver development was born called Linux Device Model. The Linux Device Model (LDM), which exposes an Object Oriented (OO)-like programming style to the kernel, provides useful utilities below:

- Supportive concept of class, to group devices that have the same functionalities;
- Communicating with the user space via *sysfs* virtual filesystem;
- Power management for handling the order in which devices should shut down;
- And hot-plugging handling mechanism about the plugging and unplugging of devices.

For the most part, the Linux Device Model code takes care of all these considerations without imposing itself upon driver authors. Direct interaction with the device model is generally handled by bus-level logic and various other kernel subsystems. So many driver authors can ignore the device model entirely, and trust it to take care of itself [12]. However, there are sometimes drivers require knowledge in Linux device model to modify specific parameter and on top of that, the model will reveal how everything connects behind the hood. Because the Linux Device Model is quite hard to understand by the top-down approach, we will use the bottom-up approach. That is provides building blocks concepts, slowly connects them to build larger concepts, and finally bring together to create an overview picture.

1.4.1 Kobject, Ksets and Subsystems

1.4.1.1. Kobject

The *kobject* is the core structure that embedded in every model inside of the Linux Device Model like atoms is made up of every material in physics. It brings an *kobject* exists only to tie a higher-level object into the device model, if it's standalone it has very little meaning. It is like inheritance in C++ but because C does not provide that function, so we have to embed struct in another struct. Let us have a look at *kobject* structure in the below figure.

```
struct kobject {
     const char *name;
     struct list_head entry;
     struct kobject* parent;
     struct kset* kset;
     struct kobj_type* ktype;
     struct dentry* dentry;
     struct kref kref;
};
```

Figure 1.19 struct kobject

- `name`: points to a string holding the name of the container (larger object which contains the *kobject*);
- `entry`: points to the list *kobject* is inserted;
- `parent`: points to this *kobject*'s parent;
- `kset`: points to the group of *kobjects* within the same structures type;
- `type`: points to the `structure` `kobj_type`;
- `dentry`: points to the dentry of the *sysfs* file associated with the *kobject*;
- `kref`: provides reference counter for the object in which it is embedded, this helps manage life cycle of the objects. At low level, the function

`kobject_get()` increments the *kobject*'s reference counter. Function `kobject_put()` decrements the reference count.

The structure that handles the *kobject* when the reference count reaches 0 is `structure kobj_type`, this structure will control what happens when the *kobject* is created and destroyed, and when attributes are read or written to. Attributes are *sysfs* files exported to the user space by *kobjects*. An attribute represents an object property that can be readable, writable, or both, from the user space. In other words, attributes map kernel data to files in *sysfs* [13].

```
struct kobj_type {
    void (*release)(struct kobject*);
    const struct sysfs_ops sysfs_ops;
    struct attribute **default_attrs;
};
```

Figure 1.20 struct kobj_type

The figure 1.20 above shows struct *kobj_type*, which has following fields:

- `release`: a callback called by the `kobject_put()` function whenever given object needs to be freed (reference count reaches 0);
- `sysfs_ops`: a set of callbacks (*sysfs* operations) called when a *sysfs* attribute is accessed;
- `default_attrs`: points to a list of `struct attribute` used as default attributes for each *kobject* of this type.

1.4.1.2. Ksets

Ksets are a collection (group) of *kobjects* related with each other. In other words, it is a higher level container class for *kobjects* and each *kset* represented by a directory in *sysfs*. Let's have a look at `structure ksets`:

```
struct kset {
    struct subsystem* subsys;
    struct kobj_type* ktype;
    struct list_head list;
    struct kobject kobj;
    struct kset_hotplug_ops* hotplug_ops;
};
```

Figure 1.21 struct ksets

- `subsys`: points to the subsystem which includes this *kset*;
- `ktype`: points to the same *kobject_type* of all *kobjects* contained in this *kset*;
- `list`: points to the its children *kobjects*' linked list;

- `kobj`: an embedded *kobject* of its own, thanks to this *kobject*, the `struct kset` can be contained in larger object, kept track of its life cycle based on reference counter of this `kobj`;
- `hotplug_ops`: points to a table of callback functions for hot-plugging.

The parent field in the *kobject* points to the containing *kset*, and the *kset*'s list points to the head of the linked list. Here is the simple *kset* hierarchy to represent the relationship between *kobject* and the *kset*:

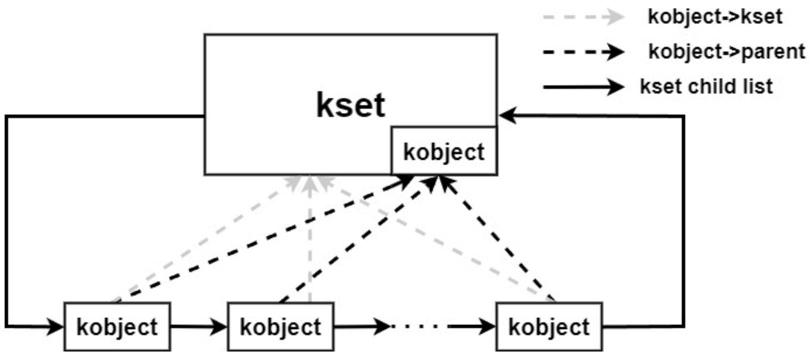


Figure 1.22 Relationship between kobject and kset

1.4.1.3. Subsystems

The collections of *ksets* called subsystems, different types of *ksets* may be included in one subsystem. Subsystems usually show up at the top of the *sysfs* hierarchy. Some example subsystems in the kernel include `block_subsys` represented as */sys/block* and `device_subsys` represented as */sys/devices* [12]. The structure of subsystem only has two fields:

```
struct subsystem {
     struct kset kset;
     struct rw_semaphore rwsem;
};
```

Figure 1.23 struct subsystem

- `kset`: an embedded *kset*;
- `rwsem`: a semaphore that protects all *ksets* and *kobjects* included in the subsystem from read-write operations.

The *subsytem* can also be embedded in a larger object, and the reference of that larger object is also the reference counter of the embedded *kobject*. The picture below demonstrates the model of *kobjects, ksets* and subsystems.

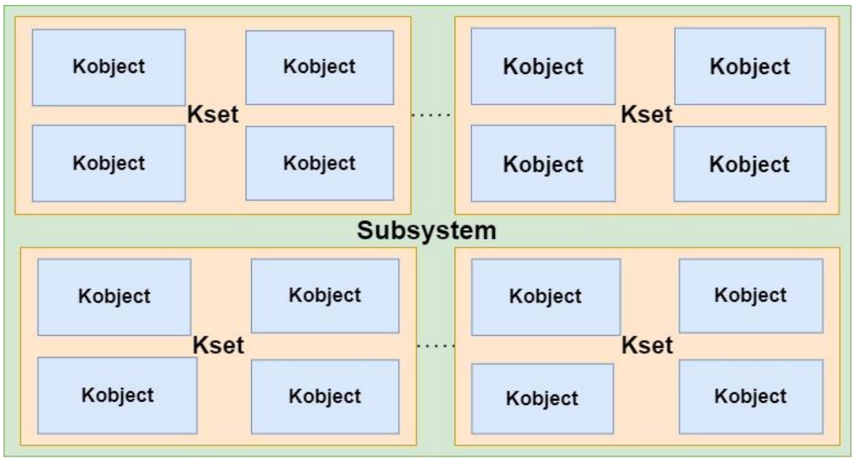


Figure 1.24 Model of kobjects, ksets and subsystems

In short, attributes represented by files which can be read or written. Lots of files contained in a directory which represents *kobject*. And collections of *kobjects* is *ksets*, which can be thought of as parent's directory. And finally group of *ksets* is subsystem, which usually appears as directories show up at the top of *sysfs* hierarchy.

1.4.2 Components of Device Driver Model

The *sysfs* filesystem is a mechanism to demonstrate a hierarchy of the system and exposes the Linux Device Driver Model by means of their *kobjects*. It provides a user interface to interact with kernel objects.

Fundamental components of Device Driver Model are illustrated in this below picture.

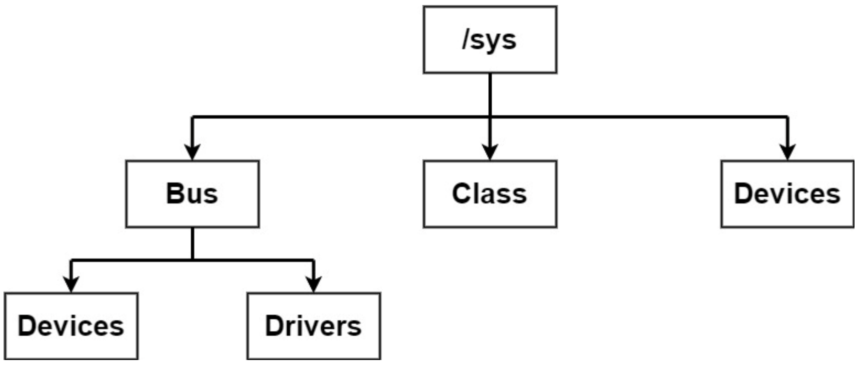


Figure 1.25 Components of Device Driver Model

Every device in the device driver model is represented by an instance of device object, let's have a look at the `struct device`:

```
struct device {
    struct list_head node;
    struct list_head bus_list;
    struct list_head driver_list;
    struct list_head children;
    struct device* parent;
    struct kobject kobj;
    char bus_id[BUS_ID_SIZE];
    struct bus_type* bus;
    struct device_driver* driver;
    void* driver_data;
    void(*release)(struct device* dev);
};
```

Figure 1.26 struct device

- `node`: Pointers for the list of sibling devices;
- `bus_list`: Pointers for the list of devices sit on the same bus types;
- `driver_list`: Pointers for the driver's list of managed devices;
- `children`: Head of the list of children devices;
- `parent`: Pointer to the parent device;
- `kobj`: *kobject* embedded in this device;
- `bus_id`: device position on the bus;
- `bus`: Pointer to the hosting bus;
- `driver`: Pointer to the controlling device driver;
- `driver_data`: Pointer to private data for the driver;
- `release`: callback function for releasing the device descriptor.

The device objects represented by the */sys/devices* directory. The devices are organized hierarchically: a device is the "parent" of some "children" devices if the children devices cannot work properly without the parent device. For example, a bridge between the PCI bus and the USB bus is the parent device of every device hosted on the USB bus. The `parent` field of the device object points to the descriptor of the parent device, the `children` field is the head of the list of children devices, and the `node` field stores the pointers to the adjacent elements in the children list. As we already know in the section platform device driver, every driver manages a list of devices, the `driver_list` field of the device object stores the pointers to the adjacent elements, while the `driver` field points to the descriptor of the device driver. For each bus type, moreover, there is a list including all devices that are hosted on the buses of the given type; the `bus_list` field of the device object stores the pointers to the adjacent elements, while the `bus` field points to the bus type descriptor [14].

The device model provides utilities to track all the drivers known to the system. Let’s analyze `struct device_driver` in the following figure:

```
struct device_driver {
     char* name;
     struct bus_type* bus;
     struct semaphore unload_sem;
     struct kobject kobj;
     struct list_head devices;
     struct module* owner;
     int(*)(struct device *) probe;
     int(*)(struct device *) remove;
     int(*)(struct device *) shutdown ;
     int(*)(struct device *, unsigned long, unsigned
long) suspend;
     int(*)(struct device *, unsigned long) resume;
};
```

Figure 1.27 struct device_driver

- `name`: Name of the device driver;
- `bus`: Pointer to descriptor of the bus that hosts the supported devices;
- `unload_sem`: Semaphore to forbid device driver unloading; it is released when the reference counter reaches zero;
- `kobj`: Embedded *kobject* in this driver;
- `devices`: Head of the list including all devices supported by the driver;
- `owner`: Identifies the module that implements the device driver;
- `probe`: Method for probing a device (checking that if it contains in the list of device drivers);
- `remove`: Method invoked on a device when it is removed;
- `shutdown`: Method invoked on a device when it is powered off;
- `suspend`: Method invoked on a device when it is put in low-power state;
- `resume`: Method invoked on a device when it is put back in the normal state.

Here, `name` is the name of the driver, which shows up in *sysfs*, `bus` is the type of bus this driver works with. `kobject` is the embedded *kobject* of this driver, `devices` is a list of all devices currently managed by this driver. The `device_driver` object includes four methods for handling hot-plugging, plug and play and power management. The `probe` method is called whenever a bus device driver discovers a device that managed by the driver. The `remove` method is invoked on a hot-pluggable device whenever it is removed; it is also invoked on every device handled by the driver when the driver itself is unloaded. The `shutdown`, `suspend`, and `resume` methods are invoked on a device when the kernel must change its power state [14].

A bus is a bridge between the processor and devices, all devices connected via a bus. Each bus is described as a `bus_type` object by the kernel. Let's have a look at the `struct bus_type`:

```
struct bus_type {
     char* name;
     struct subsystem subsys;
     struct kset drivers;
     struct kset devices;
     struct bus_attribute* bus_attrs;
     struct device_attribute* dev_attrs;
     struct driver_attribute* drv_attrs;
     int(*)(struct device *, struct device_driver *)
match;
     int(*)(struct device *, char **, int, char *, int)
hotplug;
     int(*)(struct device *, unsigned long) suspend;
     int(*)(struct device *) resume;
};
```

Figure 1.28 struct bus_type

The figure 1.28 above shows some important fields of `struct bus_type`:

- `name`: Name of the bus type;
- `subsys`: Subsystem embedded in this bus type;
- `drivers`: Group of *kobjects* of the drivers;
- `devices`: Group of *kobjects* of the devices;
- `bus_attrs`: Pointer to the object including the bus attributes and the methods for exporting them to the *sysfs* filesystem;
- `dev_attrs`: Pointer to the object including the device attributes and the methods for exporting them to the *sysfs* filesystem;
- `drv_attrs`: Pointer to the object including the device driver attributes and the methods for exporting them to the *sysfs* filesystem;
- `match`: Method for checking whether a given driver supports a given device;
- `hotplug`: Method invoked when a device is being registered;
- `suspend`: Method for saving the hardware context state and changing the power level of a device;
- `resume`: Method for changing the power level and restoring the hardware context of a device [14].

Each `struct bus_type` includes a subsystem, it contains all subsystems embedded in the `bus_type` object, like we mentioned before, subsystem can be contained in the larger object, in this case a larger subsystem object. The `subsys` subsystem represented by the */sys/bus* directory. Each bus subsystem contains two *ksets* which are drivers and devices, in other words, each bus manage drivers and devices sit on that bus, which mentioned once in the section platform device drivers. The directories of the devices already shows up in the `sysfs` filesystem under `/sys/devices`, the devices directory under the bus directories stores symbolic links pointing to the directories under `/sys/devices`. Whether there is a new device registered to the bus, it performs the `match` method to check whether a given device in the devices' list managed by the given driver. The `hotplug` method is executed when a device is being registered in the device driver model. Finally, the `suspend` and `resume` methods are executed when a device on a bus of the given type must change its power state.

Finally, class subsystem is group of devices which has the same operation functions. Each class is represented by a class object, and all class objects belong to the `class_subsys` subsystem associated with the */sys/class* directory. Each class object contains a list of `class_device` descriptors, each of which represents a single logical device belonging to the class. Logical devices are higher-view representation of physical devices which belongs to the */sys/devices* directories. The classes of the device driver model are essentially aimed to provide a standard method for exporting to User Mode applications the interfaces of the logical devices. Each `class_device` descriptor embeds a *kobject* having an attribute which is a special file named `dev`. This attribute stores the major and minor numbers of the device file that is needed to access to the corresponding logical device [14]. There is a `class` field in the `struct device` which describes the class associated with that device. The function *device_create()* initializes the device structure, and receives the class and the device as parameters, it then creates an attribute of the class, *dev*, which contains the major and the minor numbers of the device. After that, *udev*, a device manager which is responsible for managing device nodes in the */dev* directory, can read the necessary data from this attribute file and make the system call `makenod()` to create a node in */dev* directory.

1.5 Putting it all together

Now, there are enough materials to have a firm understanding of the mechanism behind the *sysfs* filesystem and how things relate to each other. We will walk through the driver register's process to summarize what we have cover in the platform device driver and Linux device driver model' section. The driver will register itself to the hosted bus through the function `driver_register()`, and as we know from the platform device driver section whenever there is a new device or a new driver registers to the bus, the bus will run the matching loop to detects the list of devices managed by the drivers which is exposed to the kernel or more precisely the bus core using the `MODULE_DEVICE_TABLE` macro and the devices node in the Device Tree based on the `compatible` field. During the compilation time, information from the `MODULE_DEVICE_TABLE` is extracted out of the driver and a file called *modules.alias* which is the list of devices managed by that driver. When a device is detected or plugged into the system, *udev* identifies the device based on its `compatible` strings from Device Tree, *udev* uses this identifier to look up entries in the *modules.alias* file to determine which module should be loaded to support the device. In short, its original purpose was to create for the hot-pluggable devices supported bus like PCI, USB,… but to obey the unifying model of device drivers, other bus drivers use this macro as well. Besides, this macro provides explicit matching which ensures the kernel can correctly bind it to the appropriate driver during initialization. Returns to the process, if there is a match, the bus invokes probe function of the driver. In the probe, there is a `device_create` function which will creates an *uevent* which is handled by the *udev*, and a device file is created for userspace-interaction. Devices, drivers, bus and attributes are created and show up as directories and files, respectively serving as an interface between user and the kernel.

The picture below illustrates the events happened when register the driver and device:

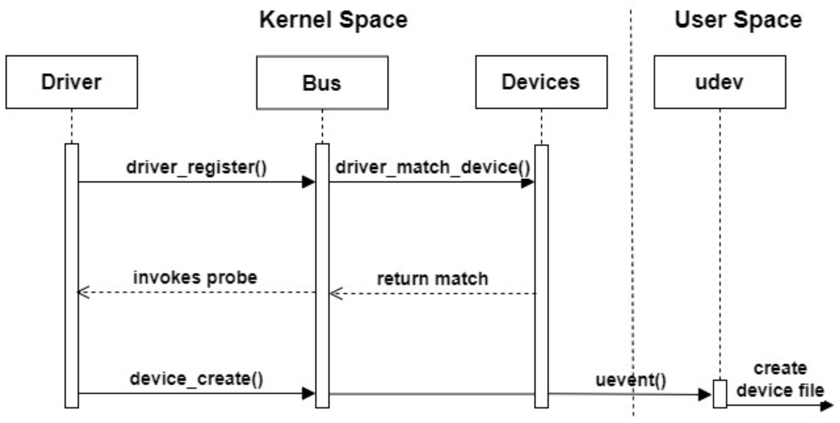


Figure 1.29 Device driver's register process

The unregister driver's process is just the reverse steps of register process. To summarize, in this chapter we provide the tools create a device driver, there are a lot of key concepts which are:

- Fundamental knowledge about kernel modules and tools to interact with them;
- The concept of system calls, how they interact with user space and some system calls mostly used in driver development;
- The concept and the skeleton of the character device drivers;

- The concept and the skeleton of the platform device drivers;
- The concept of Device Tree;
- The concept and the mechanism of the Linux Device Driver Model, which serves as a unifying model for device driver development;
- The device driver's register process, which helps to understand what happens under the hood when register new driver into the bus, and finally, the overview outline, which sums up all the things we cover from the beginning of this chapter. The overview outline is demonstrated in the following figure.

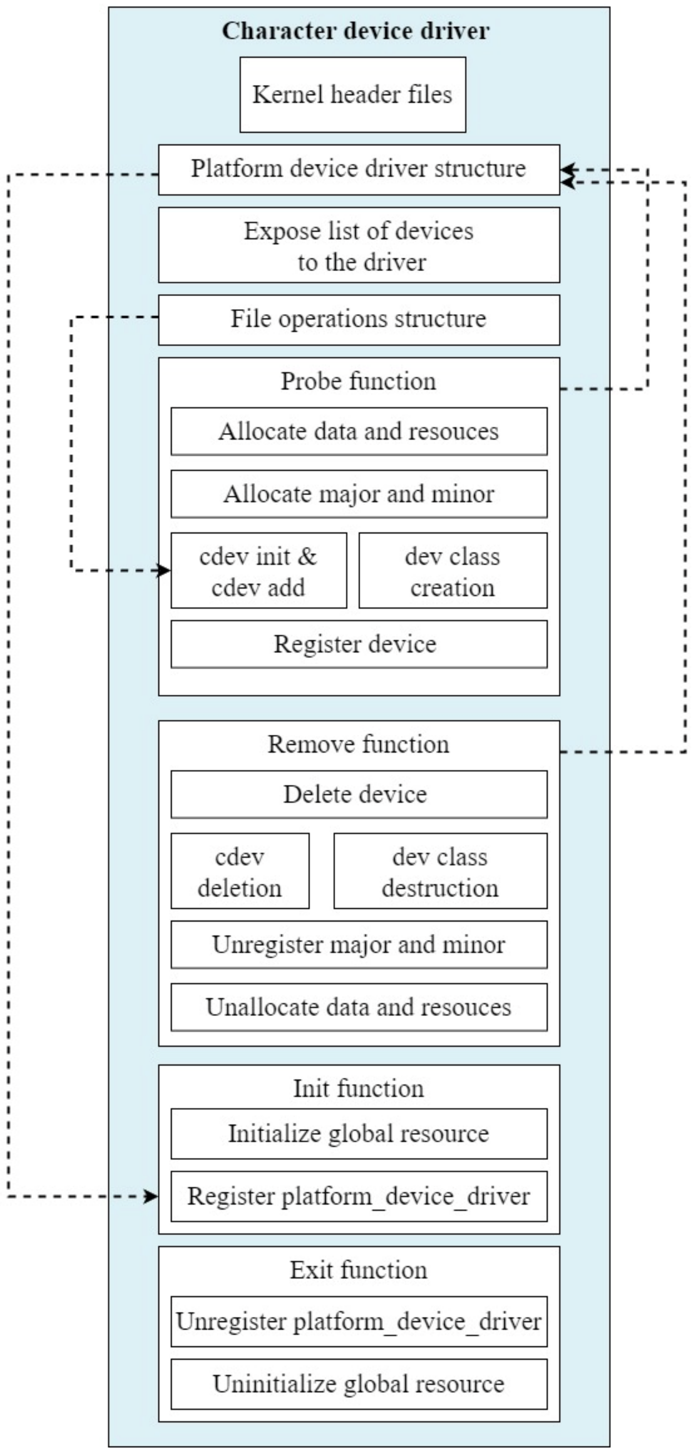

Character device driver
Kernel header files
Platform device driver structure
Expose list of devices to the driver
File operations structure
Probe function
Allocate data and resouces
Allocate major and minor
cdev init & cdev add
dev class creation
Register device
Remove function
Delete device
cdev deletion
dev class destruction
Unregister major and minor
Unallocate data and resouces
Init function
Initialize global resource
Register platform_device_driver
Exit function
Unregister platform_device_driver
Uninitialize global resource

Figure 1.30 Platform device driver skeleton

CHAPTER 2. IMPLEMENTATION OUTLINES FOR UART, SPI, AND I2C DEVICES AND RESULTS

This chapter's main focus is on how to utilize many components like device driver basis, character device drivers skeletons and platform device drivers skeletons that was mentioned in Chapter 1, and apply them to some specific devices in order to make their functional drivers. In addition to the driver basis, some particular setups and frameworks should also be taken into account for making those devices workable.

2.1 Deploying system

For programming a driver, any Linux based computers will do just fine. In the range of this thesis, we will use two different systems: a Raspberry Pi 3 Model B single-board computer, and a Beagle Bone Black.

2.1.1 Raspberry Pi 3 Model B

Raspberry Pi is the name of a series of small single-board computers produced by the Raspberry Pi Foundation. The Raspberry Pi computers are cheap, run Linux, and provide access to GPIO pins, which allows programmers to control electronic components easily and perform a variety of different tasks.

Raspberry Pi 3 Model B is the third-generation of the Raspberry Pi that came out in February 2016. Some of its technical specifications is as follows:

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU;
- 1GB of RAM;
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board;
- 40-pin extended GPIO;
- 4 USB 2 ports;
- Micro SD port for loading operating system and storing data;
- Micro USB power source up to 2.5A.

2.1.2 Beagle Bone Black

Beagle Bone Black is a low-cost, high-expansion, communication-supported development platform for developers and hobbyists. Beagle Bone Black is designed to address the Open-Source Community, early adopters, and anyone interested in a low-cost ARM Cortex-A8 based processor. It has been equipped with a minimum set of peripherals to allow the user to experience the power of the processor and offers utilities to perform many tasks suited for the embedded applications.

- 1GHz ARM Cortex-A8;
- 512MB SDRAM;
- 4GB Onboard Flash;
- 10/100M Ethernet (RJ45);
- 65-pin extended GPIO;
- MicroSD port for insert MicroSD Card, which stores OS and data;
- Mini USB or DC Jack for power source.

2.2 Deployment on UART devices

2.2.1 UART

UART (Universal Asynchronous Receiver-Transmitter) is a peripheral device for asynchronous serial communication in which the data format and transmission speeds are configurable. The data bits are sent one by one, from the least significant to the most significant one. The data is also framed by start and stop bits so that the communication channel's timing is handled precisely. The two signals of each UART device are named the transmitter (TX) and the receiver (RX). This device's transmitter is connected to the other device's receiver, and the same backward, which is depicted in the following Figure 2.1.

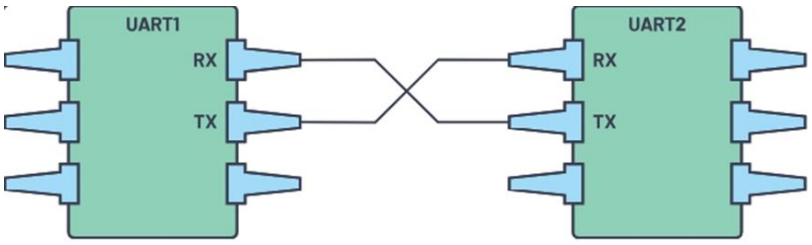
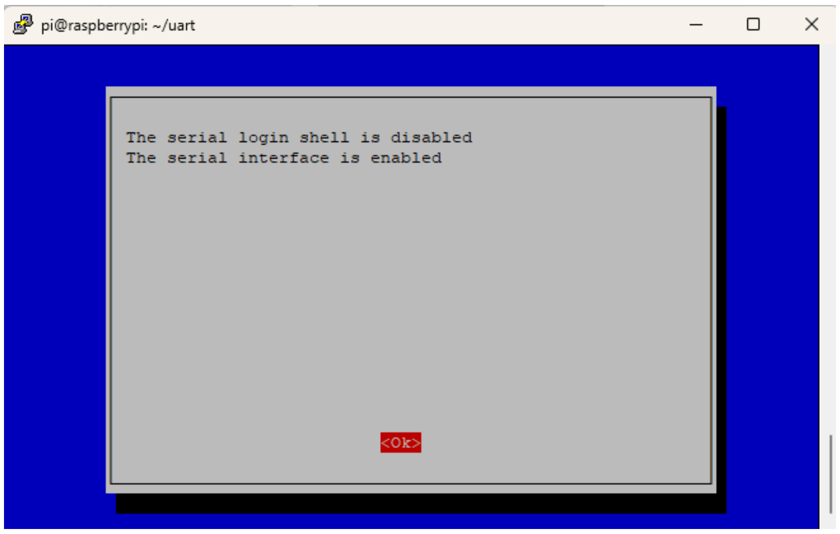


Figure 2.1 UART devices depiction

A UART data frame usually consists of five elements:

- Idle state: Usually logic high level (1);
- Start bit: 1 bit, logic low level (0);
- Data bits: 5 to 9 bits;
- Parity bit: 0 to 1 bit placed after all the data bits, that is used to tell if any data has changed during the transmission;
- Stop bit(s): 1 to 2 bits, logic high level (1).

"There are two types of UART available on the Raspberry Pi – PL011 and mini UART. The PL011 is a capable, broadly 16550-compatible UART, while the mini UART has a reduced feature set [15]." On the Raspberry Pi 3, the primary UART is UART1 which is the mini UART, and the secondary UART is the PL011. The primary UART is by default selected to GPIO 14 (TX) and GPIO 15 (RX). Setting up UART can be done by using the *sudo raspi-config* command, entering the Interface Options, and enabling the Serial Port. One key thing to remember is that if the serial login shell is not disabled, this will result in a few bugs with transmitting data. The final result after setting should look something like in Figure 2.2 below.



```
The serial login shell is disabled
The serial interface is enabled
```

Figure 2.2 Serial port setup result

2.2.2 FPC1020A Capacitive Fingerprint Sensor Module

FPC1020A Capacitive Fingerprint Sensor Module is a fingerprint detection module for fingerprint recognition, specifically designed and optimized for microcontrollers and many mobile devices. It is a compact size module with low energy consumption and high reliability.

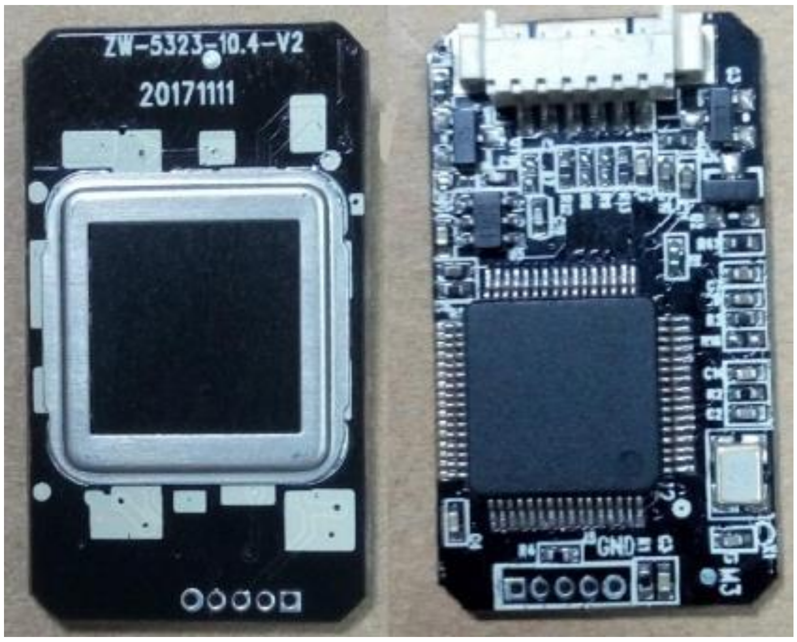


Figure 2.3 FPC1020A Capacitive Fingerprint Sensor Module [16]

2.2.2.1. Module features

This module has the following features:

- Fingerprint capacity of 150 fingerprints;
- FPC1020 CMOS fingerprint touch sensor;
- The sensor resolution of 508 dpi;
- 160 x 160 pixels with 8 bit depth;
- One-to-many and one-to-one fingerprint identification;
- Fingerprint extraction time and identification time < 0.45 seconds;
- Fake rate < 0.0001%;
- Refuse rate < 0.001%;
- Operating voltage of 3.3V;
- Working current < 50 mA;
- Sleeping current < 3µA;
- Communication interface: UART or USB;
- Baud rate: 19200 bps.
- Operating temperature: -10ºC ~ 60ºC;
- Operating humidity: 20% ~ 80%.

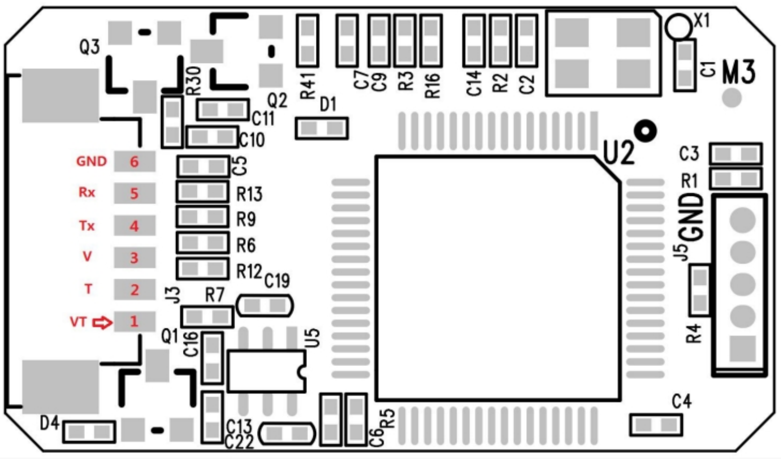


Figure 2.4 FPC1020A module's circuit and pinout diagram [16]

2.2.2.2. Pinout description

The details of the module's pinout depicted in Figure 2.4 is in Table 3.1 below.

Table 2.1 FPC1020A module pinout description [16]

Pin name	Pin type	Description
VT	POWER	Power supply for finger touch detecting function, voltage range 2.5V ~ 5.5V, average 3.3V
T	OUTPUT	Output high (3.3V) when finger touch

		is detected, otherwise output low (0V)
V	POWER	3.3V power supply for the module
Tx	OUTPUT	Transmitter
Rx	INPUT	Receiver
GND	GROUND	Ground

2.2.2.3. Module's data protocol

The FPC1020A module has its protocol to format data for sending to or receiving from other devices as in the following table.

Table 2.2 FPC1020A protocol format [16]

Byte	1	2	3	4	5	6	7	8
Command	0xF5	CMD	P1	P2	P3	0	CHK	0xF5
Response	0xF5	CMD	Q1	Q2	Q3	0	CHK	0xF5

Table 3.2 shows that the data frame consists of 8 bytes, in which:

- The start byte and stop byte must be 0xF5;
- CMD is the command number to send to the module;
- P1, P2, and P3 are the command dependent parameters;
- Q1, Q2, and Q3 are the returning parameters, specifically Q3 is used to return a value that represents if the operation has succeeded or not (the values for Q3 is defined in Appendix A.1. section);
- And CHK is the checksum byte, which is calculated by performing the logical XOR operation every values from the second to the sixth byte.

In cases when there is a need to send or receive more data than just 8 bytes, the data frame is then split into data head and data packet. The formats of both of them are described in the tables below.

Table 2.3 FPC1020A data head [16]

Byte	1	2	3	4	5	6	7	8
Command	0xF5	CMD	Hi(Len)	Low(Len)	0	0	CHK	0xF5
Response	0xF5	CMD	Hi(Len)	Low(Len)	Q3	0	CHK	0xF5

Table 2.4 FPC1020A data packet [16]

Byte	1	2 ~ Len + 1	Len + 2	Len + 3
Command	0xF5	Data	CHK	0xF5
Response	0xF5	Data	CHK	0xF5

The data head in Table 3.3 is sent (or received) first, and then comes the data packet in Table 3.4. In both tables:

- CMD and Q3 is the same as mentioned above;
- Len is the length of the data packet, in the form of a 16-bit number (2 bytes);
- Hi(Len) is the high 8 bits of the data length;
- Low(Len) is the low 8 bits of the data length;
- And CHK is the checksum byte, which is calculated by performing the logical XOR operation every values from the second to the sixth byte in the case of data head, or all the bytes except the first and the last two bytes in the case of data packet.

Command numbers defined by the module can be found in the Appendix A.1. section.

Each command and how to retrieve the response data should follow the previously mentioned module's data protocol requirements and the method to implement those behaviours in a Linux kernel driver.

2.2.3 Serdev framework

Implementing UART communication in a Linux kernel driver is considered such a "weird" case when comparing to other types of protocols. Traditionally, the TTY layer (Figure 2.5) handles the communication between UART devices and user programs.

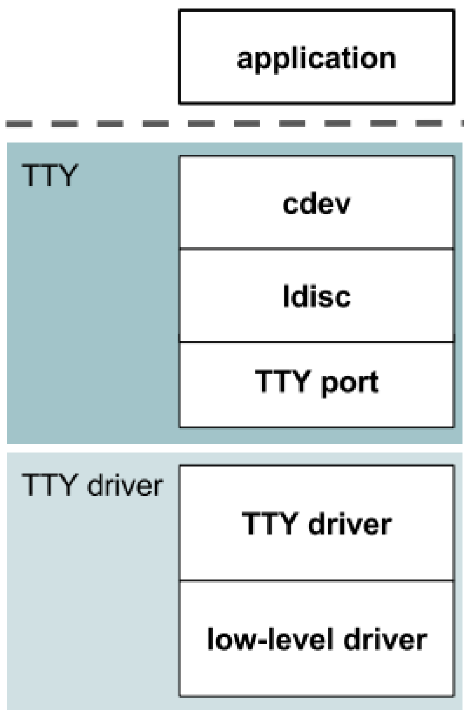


Figure 2.5 TTY layer [17]

As portrayed in Figure 2.5, the communication from the user application is first handled by the *cdev*, and then the *ldisc* which is the *line discipline* that is in charge of the input/output processing, error handling and status monitoring. After that, it is passed down to the lower TTY drivers and other low-level drivers that directly control the hardware and pseudo-hardware. This traditional approach has many drawbacks, one of which is the fact that it does not fit to the Linux Driver Device Model. In addition, the line disciplines have some limitations like handling sideband signals, power control, and firmware loading.

As a result, since the 4.12 version of the kernel, the *serial device bus* (often called *serdev*) has been introduced to improve the existing issue. The *serdev* allows a UART device to be attached without having to work with the troublesome TTY layer.

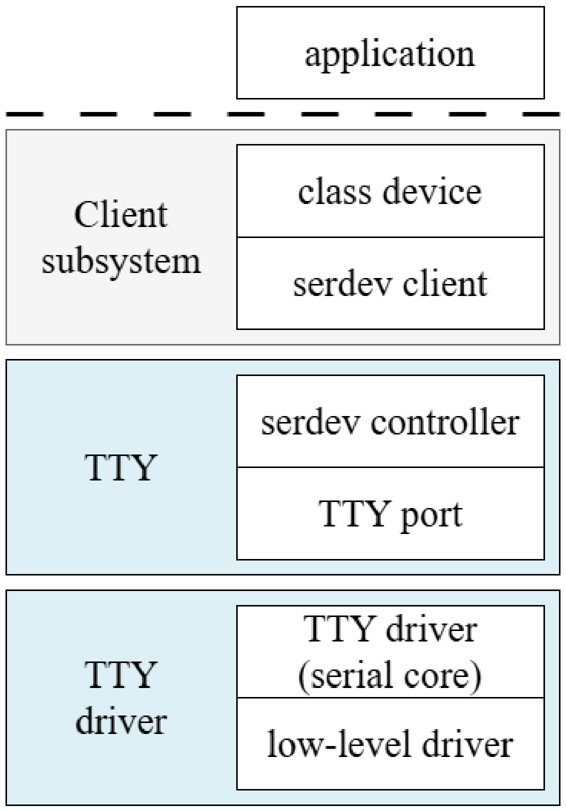


Figure 2.6 Serdev framework

As shown in Figure 2.6, the serdev framework has the following components:

- Serdev controller;
- Serdev class device (also known as client device or slave device);
- Serdev TTY-port controller:
    - Only in-kernel controller implementation;
    - Registered by TTY driver when client is defined;
    - Clients are described by firmware (Device Tree or ACPI) [17].

At this point, it can be seen that the *serdev driver* shares many similarities to the *platform device driver* and follows the Linux Driver Device Model mentioned in Chapter 2.

```
int      serdev_device_open(struct serdev_device *);
void     serdev_device_close(...);
unsigned serdev_device_set_baudrate(...);
void     serdev_device_set_flow_control(...);
int      serdev_device_write_buf(...);
void     serdev_device_wait_until_sent(...);
void     serdev_device_write_flush(...);
int      serdev_device_write_room(...);
int      serdev_device_get_tiocm(...);
int      serdev_device_set_tiocm(...);
```

Figure 2.7 Serdev driver interface functions [17]

The serdev framework provides many different functions that are used in UART channel setups and transferring data, as shown in Figure 2.7. In addition, there are also two callbacks for manipulating workflows in the driver in the figure below.

```
struct serdev_device_ops {
        int (*receive_buf)(struct serdev_device *,
                                const unsigned char *, size_t);
        void (*write_wakeup)(struct serdev_device *);
};
```

Figure 2.8 Serdev driver interface callbacks [17]

As shown in Figure 2.8, the `receive_buf` pointer can be assigned with a function that implements the behaviours of the driver when it receives new data. It is the same with `write_wakeup`.

```
static struct serdev_device_driver slave_driver = {
    .driver = {
        .name           = "serdev-slave",
        .of_match_table = of_match_ptr(slave_of_match),
        .pm             = &slave_pm_ops,
    },
    .probe  = slave_probe,
    .remove = slave_remove,
};

module_serdev_device_driver(slave_driver);
```

Figure 2.9 Example serdev driver [17]

Figure 2.9 is an example of a serdev driver, in which, after assigning its members properly, the `module_serdev_device_driver()` function is called to register the driver. This function is an alternative to the `module_init()` and `module_exit()` macros of the kernel so using either of which methods is fine.

2.2.4 Implementation outline for the FPC1020A module

2.2.4.4. Driver requirements

For the case of the FPC1020A module, we want the driver to have the following requirements:

- Captured image should be identical to the fingerprint;
- Commands should be performed under 1 second, with the exception of the process of extracting fingerprint should be under 3 seconds;
- And other module features that was mentioned in the above section.

2.2.4.5. Implementation outline

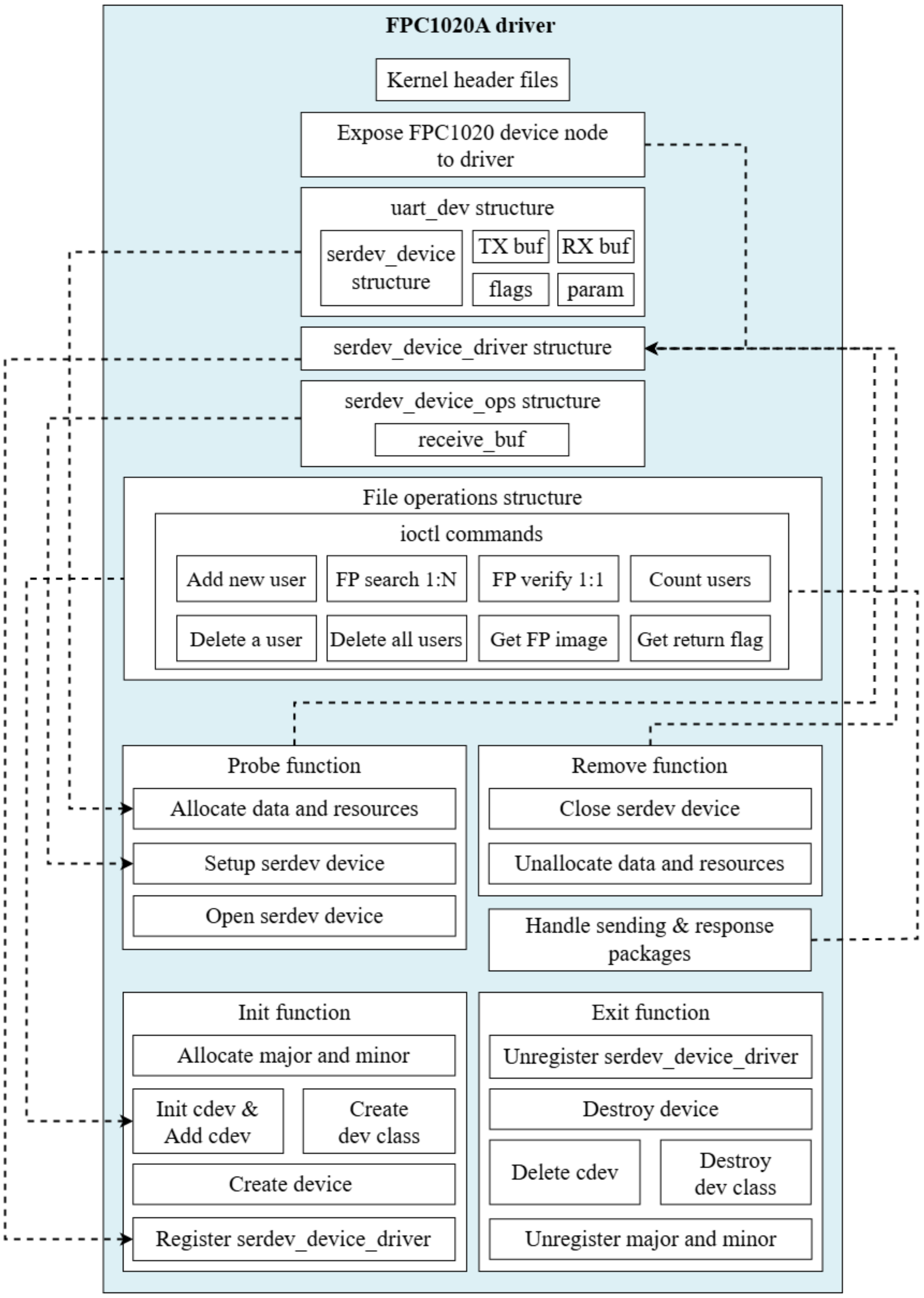


Figure 2.10 FPC1020A driver outline

Figure 2.10 is an overview of many components in the FPC1020A driver. In this section, we will take a detail look on these components. The steps to implement them is shown as follows.

- First, we have to set up our developing environment by enabling the Serial Ports on GPIO 14 and 15 of the Raspberry Pi and restarting the system. Once the system is restarted, install the kernel headers by using the `sudo apt install raspberrypi-kernel-headers` command.

- To connect the FPC1020A module with the Raspberry Pi, we will wire as the following figure.

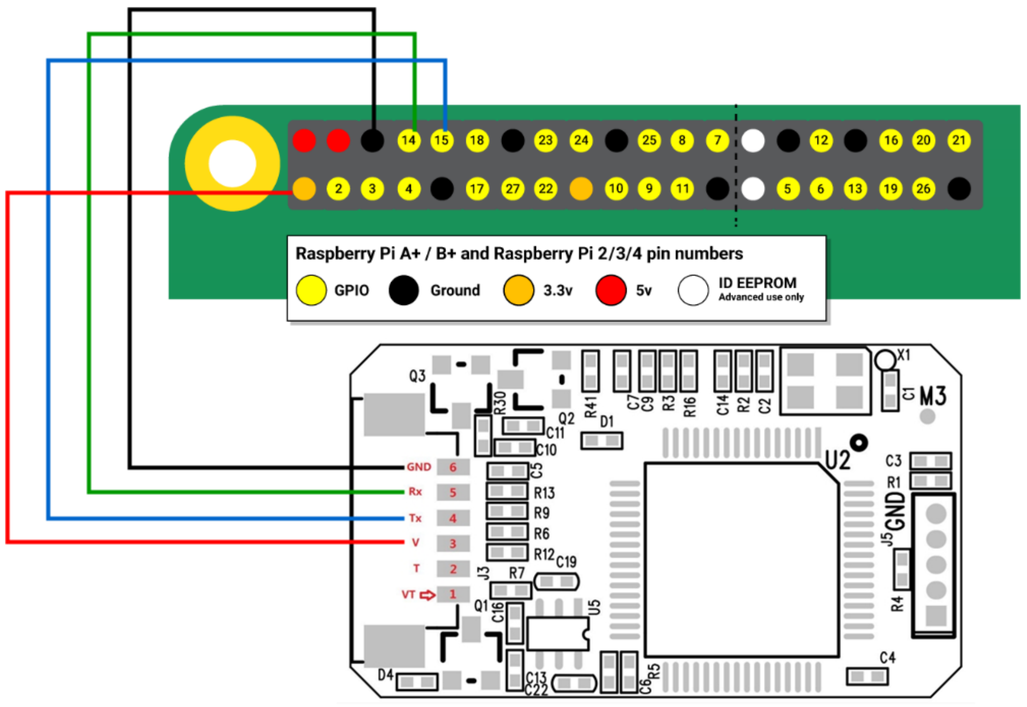


Figure 2.11 Wiring diagram of FPC1020A with Raspberry Pi

- Next, we create the driver code file and declare a `struct uart_dev` which contains `struct serder_device` and some other important members like TX and RX buffers, count flag, return flag, input or output parameters, etc. By doing this way, we can create an object that represents our hardware device in the driver with its properties, like OOP. Most of the module's data should be manipulated by this structure.
- We will declare `probe`, `remove`, `init`, `exit` functions, a `struct file_operations`, a `struct serdev_device_ops` with an `receive_buf` function for receiving data from the module, and a `struct serdev_device_driver`. The `receive_buf` function is a callback function that is called whenever a new character is received, so it should be implemented with a *waitqueue* in order to wake up the current process when the data is fully received. This function should also copy every character sent from the module to the TX buffer (the module's TX is the Pi's RX) inside the `struct uart_dev`.
- In order to expose the device to the driver, before calling the `MODULE_DEVICE_TABLE` macro, we must create a `struct of_device_id` that list our devices' names, but in this case we only have

one device so there is just one. The creation of the FPC1020A device node for the Device Tree on the Raspberry Pi can be performed by Device Tree Overlay so that there is no need to rebuild the whole Device Tree. Details of the `of_device_id` code can be found in the Appendix A.2. section.

- Inside the `probe` function, an instance of the `struct uart_dev` should be allocated on the heap by using `devm_kzalloc()` function, which is the same as `kmalloc()` but will initially zero out the memory range and automatically free it when we unload our device. Then, we must setup serial communication (UART) inside this function with some other functions provided the *serdev* framework. The driver's data is set using the `serdev_device_set_drvdata()` function; the device operations structure is assigned using the `serdev_device_client_ops()` function; `serdev_device_set_baudrate()` function must be used to set the baud rate to 19200 bps; `serdev_device_set_parity()` should be called with `SERDEV_PARITY_NONE` since there is no parity bit. TX and RX buffers' memory allocation should also be called inside the `probe` function, along with some other initial parameters. Details of the probe function can be found in the Appendix A.2. section.
- The `remove` function should first close the serdev device when not in used anymore and free what is allocated by the `probe` function earlier.
- Once the `probe` and `remove` function is ready, they should be linked to the `struct serdev_device_driver` along with some information about the driver and the `of_match_table` that is defined earlier.
- Next, we will implement the file operations for handling system calls from the user space. The `open`, `release`, `read` and `write` system calls won't do much but noticing the kernel about them being called. However, the `ioctl` system calls is way more crucial. The ioctl function should be implemented with a standard switch-case structure in C code as there are many different commands like adding new user, deleting users, performing fingerprint identification and so on. The method for implementing ioctl calls was shown in details for both user space and kernel space in the ioctl section in Chapter 2. There are only two things that must be remembered: for calls that require data being transferred from user to kernel space and backward, the `copy_from_user()` and `copy_to_user()` macros should be used; and if the commands require larger memory range (for example, in this case, for the receiving buffers for storing fingerprint image with a large size), the krealloc() macro can be called to reallocate to a larger range of memory.
- The driver should handle the sending package and response package in functions that are independent to the ioctl calls. This will make to ioctl codes

won’t be too complicating. The sending function for sending data to the module should have the `serdev_device_write_buf()` function. On the other hand, the checking response function should implement the waitqueue in order to wait for the data to be fully received before checking the received package.

- After, we will perform some procedures for the `init` and `exit` functions. Since we only have one piece of device in this case, some allocations can be performed in the `init` function rather than in the `probe` function. For the `init` function, first, we will do allocation for the major and minor numbers. Then, we will initialize the `cdev` for the FPC1020A along with the file operations structure defined above, and add it. Next, we will create the device class and create the device so that it appears under the */dev* directory for the user applications to access. After that, we will call the `serdev_device_driver_register()` function and pass the `struct serdev_device_driver` instance that was initialized earlier in order to register the driver with the kernel. For the `exit` function we will undo everything that the init has done, the same like what we do with the `remove` function. Once the init and exit functions are finished, we assign them with the `module_init()` and `module_exit()` macros to give the driver its entry and exit points.
- Before compiling the driver, we must first create a Device Tree Overlay for the driver to be able to match with the device. Details of the Device Tree Overlay file can be found in the Appendix A.3. section. After the .dts overlay file is compiled with the *dtc* tool, the resulting *.dtbo* file should be put in the */overlays* folder in the */boot* directory for when rebooting the system, the overlay is loaded.
- Finally, we will write the Makefile script and *make* the driver. Details of the Makefile script is in the Appendix A.4. section.

2.3 Deployment on I2C devices

2.3.1 I2C

The I2C stands for Inter-Integrated Circuit, it was invented by Philips Semiconductor in 1982 for a bidirectional synchronous serial bus communication. I2C is widely used for short distance communication, providing simple and robust communication between the Peripheral devices and the microcontroller. It uses two wires which is SDA for data transmission and SCL for synchronization data transmission to and from the target devices. The following figure describes the I2C protocol architecture.

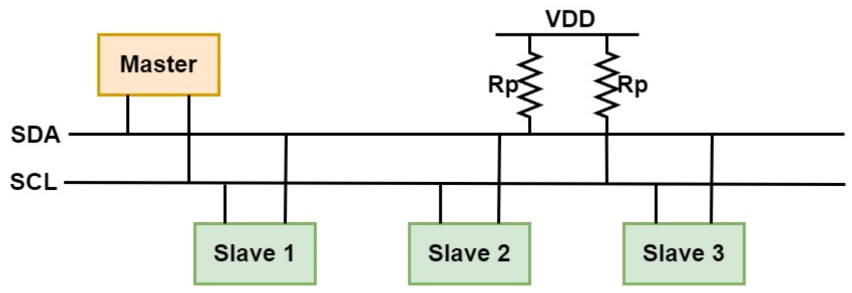


Figure 2.12 I2C protocol architecture

I2C allows for a multi-master mode although this mode is not widely used, both SDA and SCL are open drain/open collector, meaning that each of these can drive its output low but neither of them can drive its output high without having pull-up resistors. I2C is a half-duplex protocol, which means one at a time either master or slave can send data.

2.3.2 AT24C256 EEPROM

2.3.2.1. Module features

EEPROM is non-volatile memory that can be erased and reprogrammed by the user. There is also a Flash memory which is a form of EEPROM built for high speed and high density. However, the greatest difference is that Flash reads and writes data by block of sector while EEPROM reads and writes data by byte.

The AT24C256 EEPROM is manufactured by Atmel, the EEPROM provides 262,144-bits of Serial Electrically Erasable and organized as 32,768 words of 8-bits each. There are some features of the device based on the datasheet [18]:

- Low-voltage and Standard-voltage Operation:
  - VCC = 1.7V to 5.5V;
- Internally Organized as 32,768 x 8;
- 2-wire Serial Interface;
- Schmitt Trigger, Filtered Inputs for Noise Suppression;
- Bidirectional Data Transfer Protocol;
- 400kHz (1.7V) and 1MHz (2.5V, 2.7V, 5.0V) Compatibility;
- Write Protect Pin for Hardware Protection;
- 64-byte Page Write Mode;
- Self-timed Write Cycle (5ms Max);
- High Reliability:
  - Endurance: 1,000,000 Write Cycles;
  - Data Retention: 40 years;
- Lead-free/Halogen-free Devices Available;
- Green Package Options (Pb/Halide-free/RoHS Compliant):
  - 8-lead JEDEC SOIC, 8-lead TSSOP, 8-pad UDFN, and 8-ball VFG;

- Die Sale Options: Wafer Form, Waffle Pack, and Bumped Wafers.

The image of the device is illustrated in the figure below.

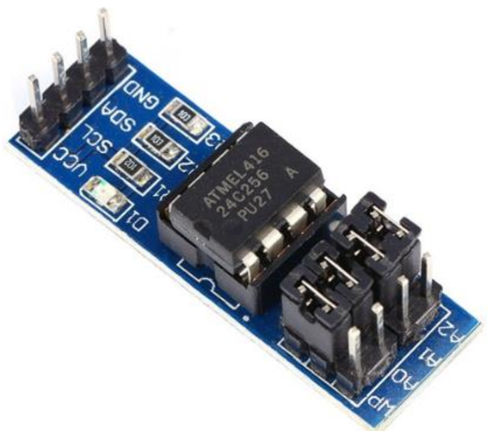


Figure 2.13 AT24C256 EEPROM

2.3.2.2. Pinout description

The pin configuration is provided in the following table.

Table 2.5. Module AT24C256 EEPROM pinout description [18].

Pin	Function
A0	Address Input
A1	Address Input
A2	Address Input
GND	Ground
SDA	Serial Data
SCL	Serial Clock Input
WP	Write Protect
VCC	Device Power Supply

2.3.2.3. Module's data protocol

One of the most important frames needed attention is the frame of the addressing device. The 256K EEPROM requires an 8-bit device address word following a START condition to enable the chip for a read or write operation. The device address word consists of a mandatory one, zero sequence for the first four most significant bits as shown in the figure below. This is common to all 2-wire EEPROM devices.

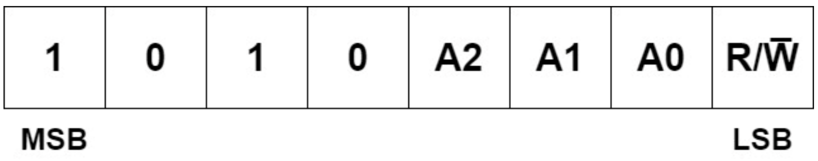


Figure 2.14 Frame of device address

In the Figure 2.14 above, the three address device bits A2, A1, and A0 allow as many as eight devices on the same bus. These bits must be compared to their corresponding hardwired input pins. When these pins are left floating, the A2, A1, and A0 pins will be internally pulled down to GND. The eighth bit of the device address is the read/write operation select bit. A read operation is chosen if this bit is HIGH, and a write operation is chosen if this bit is LOW. The data format for the I2C protocol is shown in the below picture.

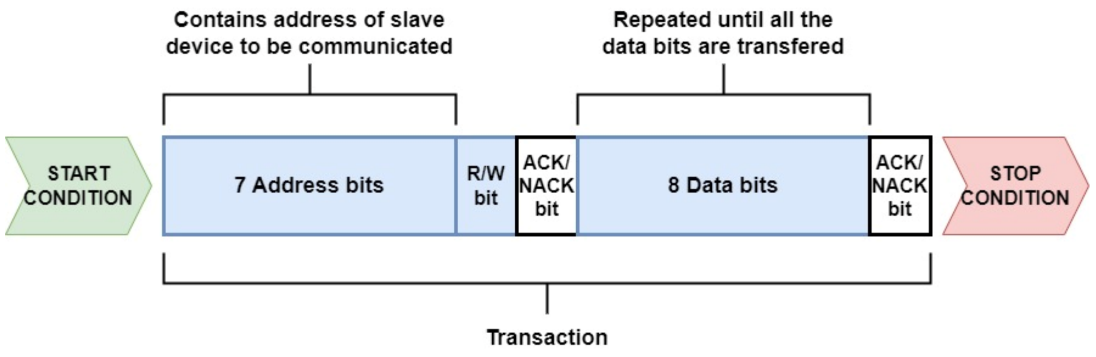

Contains address of slave device to be communicated
Repeated until all the data bits are transfered
START CONDITION
7 Address bits
R/W bit
ACK/ NACK bit
8 Data bits
ACK/ NACK bit
STOP CONDITION
Transaction

Figure 2.15 Format of message data

The main functionalities of the EEPROM follow the format data of the I2C protocol and can be found in the datasheet [18].

2.3.3 I2C framework

The I2C subsystem framework in Linux is demonstrated in the following figure.

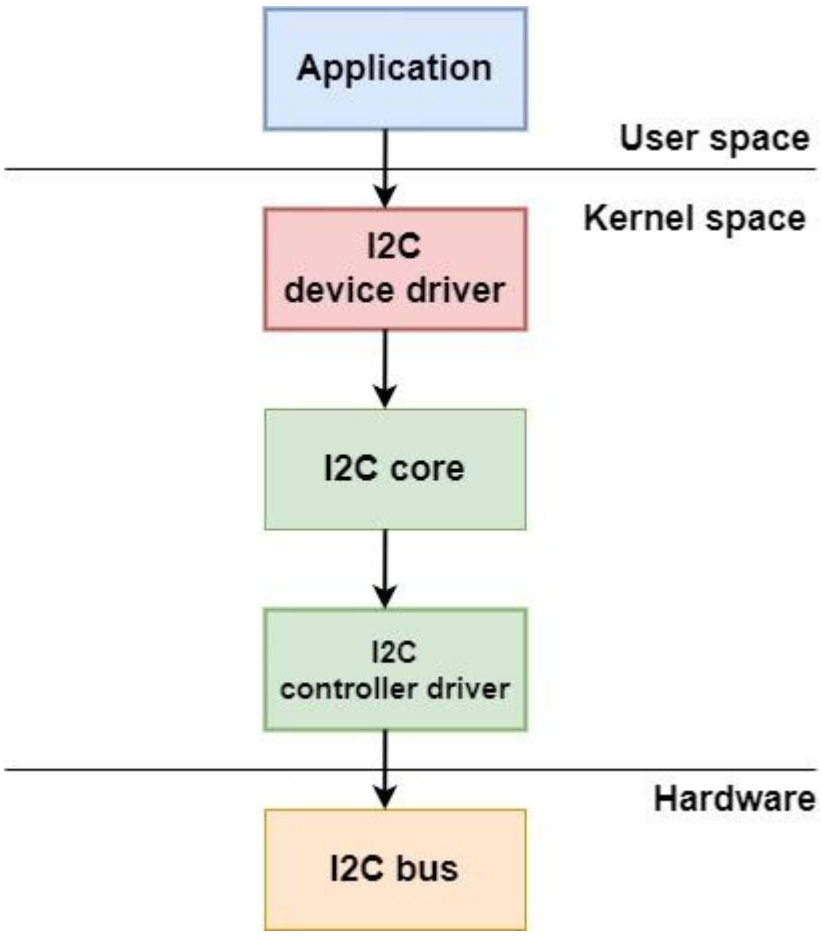


Figure 2.16 I2C subsystem framework in Linux

There are 3 main components of the I2C driver framework in the kernel.

- I2C core: A central component that provides a unified API for both I2C controllers driver and I2C device drivers. It manages the registration and unregistration of I2C bus and I2C devices. It provides functions for transferring data between the I2C master, which is the I2C bus, and the I2C slave, which is the device.
- I2C controller driver: A driver manages a specific I2C bus because a bus is also a device. It abstracts the hardware-specific details of the I2C controller. The I2C controller driver handles the initialization and communication with the I2C controller hardware. It implements functions to send and receive data on the I2C bus. This driver is usually written by the manufacturer of the microcontroller.
- I2C device driver: This is a component that most device driver developers need to pay attention to. The driver manages a specific I2C device called a slave connected to the I2C bus. It abstracts the details of the specific I2C device, providing device-specific functionality to the higher layers of the kernel.

The I2C driver exposes a lot of functions to support the data transfer between the master and the slaves. One of them is the function `i2c_transfer()` which can send multiple messages. The function is illustrated in the below picture.

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

Figure 2.17 i2c_transfer function

The parameters depicted in the Figure 2.17 above are:

- `adap`: *struct i2c_adapter* or *struct i2c_client* of the i2c device;
- `msg`: The message that we want to send to the I2C bus, the message has the following fields:
    - o *addr*: slave's address;
    - o *flags*: message's flags;
    - o *len*: msg length;
    - o *buf*: pointer to `msg` data.
- `num`: Number of messages being transferred.

It can be seen that the I2C driver shares lots of similarities with the platform device driver. The below figure shows an example of `struct I2C_driver`.

```
static struct i2c_drive foo_driver = {
     .driver = {
     .name = "foo",
     },
     .id_table = foo_idtable,
     .probe = foo_probe,
     .remove = foo_remove,
}
```

Figure 2.18 Example of struct i2c_device

In the next section, we will apply the character device driver skeleton provided at the end of the Chapter 2, EEPROM's hardware datasheet, and the I2C Linux drivers facilities to write a device driver for the AT24C256 EEPROM.

2.3.4 Implementation outline for the AT24C256 EEPROM

2.3.4.4. Driver requirements

With module AT24C256 EEPROM, we require the following functions:

- Byte write and current read operations;
- Page write and Sequential read.

2.3.4.5. Implementation outline

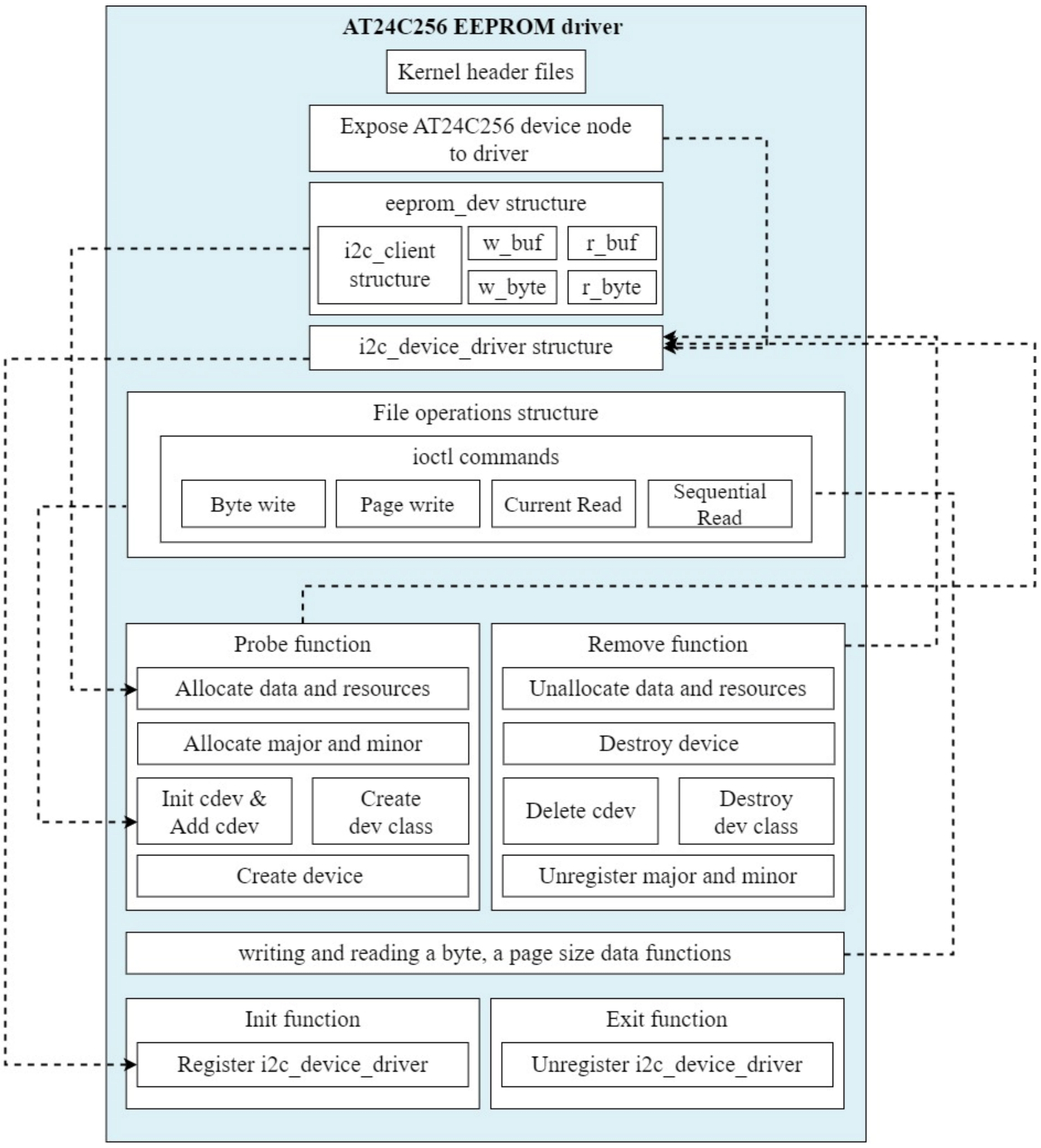


Figure 2.19 AT24C256 EEPROM driver outline

Figure 2.19 is an overview of many components in the AT24C256 EEPROM driver. In this section, we will take a detailed look at these components. The steps to implement them are shown as follows.

- To connect the module AT24C256 EEPROM with the Beagle Bone Black Board, we connect pins as the following table.

Table 2.6. Pin connection of AT24C256

AT24C256	**Beagle Bone Black**
VCC	5V
GND	Ground

SCL	P9_19
SDA	P9_20

- The next step is to allocate resources for the module by creating a device node in the file *am335x-boneblack.dts*. The details of the device node in the Device Tree can be found in Appendix A.7. section.
- Now, we create the driver code file and declare a `struct at24c256` which contains `struct i2c_client` and some other important members like write buffer, read buffer, read byte, and write byte. We want to embed those properties inside the `struct at24c256` so that this will behave like object in OOP.
- We then declare `probe`, `remove`, `init`, `exit` functions, a `struct file_operations`, and other functions needed to perform desired tasks.
- To expose the device to the driver, before calling the `MODULE_DEVICE_TABLE` macro, we must create a `struct of_device_id` that lists our devices' names, but in this case, we only have one device so there is just one. Details of the `of_device_id` code can be found in the Appendix A.7. section.
- Inside the `probe` function, an instance of the `struct at24c256` should be allocated by using `kzalloc()` function, which is the same as `malloc()` in the user space, we need to remember to free it when we unload our device otherwise we will get memory leak. We now create a character device file to interact with the user by allocating the major, minor numbers, create class, register device file with the file operations structure. The driver's data is set using the `i2c_set_client()` function. Details of the `probe` function can be found in the Appendix A.10. section.
- The `remove` function should unallocate what is allocated by the `probe` function earlier like class, major, minor numbers, and memory allocated for `struct at24c256`.
- Once the implementation of the `probe` and `remove` function is done, they need to register with the `struct i2c_driver` along with some information about the driver and the `of_match_table` that is defined earlier.
- The next step is to implement the file operations for handling system calls from the user space like `open`, `release`, `read`, and `write`. The EEPROM can be implemented only using `write` and `read` function without the need of `ioctl` function since its main function is just read from and

write to the memory segment. But to test different write and read operations we still use the `ioctl` function. The building block for all the functions to transmit data is the function `i2c_transfer`, which can send multiple messages to the bus, the messages can be customized through the `struct i2c_msg`. Finally the `ioctl` function is used to perform read and write a byte, read and write a page functions. To copy the data sent from the user to the kernel space and copy the data sent from the kernel to the user space, we use the function `copy_from_user` and the function `copy_to_user`.

- After, we implement the `init` and `exit` functions. In this function, we need to register and unregister the `struct i2c_driver`. We just simply need to pass the `struct i2c_driver` instance to the macro `module_i2c_driver()` in order to register and register the driver with the kernel.
- Finally, we will write the Makefile script and *make* the driver. Details of the Makefile script is in the Appendix A.12. section

2.4 Deployment on SPI devices

2.4.1 SPI

SPI is a protocol for synchronous serial communication. SPI has an advantage over I2C and UART is that its data can be transferred without interruption. With I2C and UART, data is sent in packets, which is limited to a specific number of bits. Moreover, there are start and stop conditions that define the beginning and end of each packet, so the data is interrupted during transmission. Devices communicating via SPI are in a master-slave relationship. The master is the controlling device (usually a microcontroller), while the slave (usually peripherals devices) takes instruction from the master. The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave. The relationship is illustrated in the below figure.

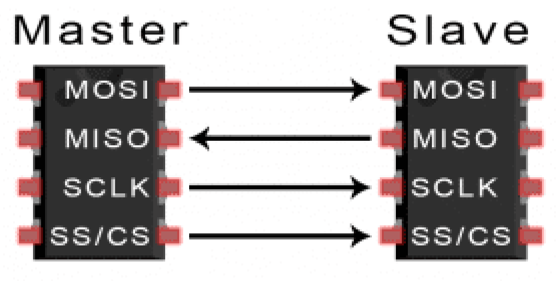


Figure 2.20 SPI master and slave

Based on the Figure 2.20 above, there are four main pins that related to SPI:

- MOSI (Master Output/Slave Input): Line for the master to send data to the slave;

- MISO (Master Input/Slave Output): Line for the slave to send data to the master;
- SCLK (Clock): Line for the clock signal;
- SS/CS (Slave Select/Chip Select): Line for the master to select which slave to send data to.

2.4.2 LCD NOKIA 5110

The module LCD Nokia 5110 is one of the most nostalgic products and is still effective for a surprisingly wide range of applications. It was originally designed for the Nokia 5110 cell phone. It features a Philip PCD8544 interface chip and has a reasonably impressive resolution of 84x48 pixels despite the age of its technology. The module LCD Nokia 5110 is illustrated in the following figure.

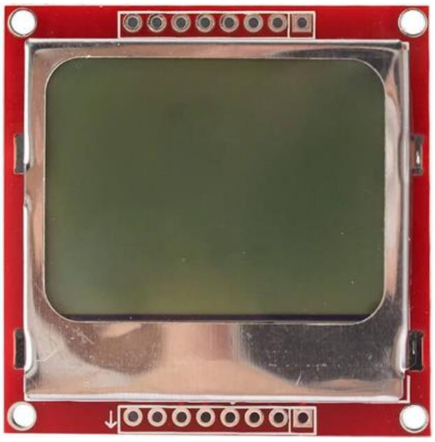


Figure 2.21 Module LCD Nokia 5110

2.4.2.1. Module features

The module has the following features:

- 48 row, 84 column outputs;
- Display data RAM 48 x 84 bits;
- Logic supply voltage range V_{DD} to V_{SS}: 2.7V to 3.3V;
- Operating voltage: 2.7V to 3.3V;
- Operating current: 6 mA;
- Serial interface maximum 4.0 Mbits/s;
- Temperature range: -25 to 70ºC;
- Low power consumption, suitable for battery-operated systems.

2.4.2.2. Pinout description

The pinout of the module is shown in the below table.

Table 2.7 **Module LCD Nokia 5110 pinout description.**

Pin	Function
VCC	3.3V
GND	Ground
SCE	Serial Chip Selection
RST	Reset
D/C	Data/Commands Choice
DIN	Serial Data Line
CLK	Serial Clock Speed
LED	Backlight Control Terminal

2.4.2.3. Module's data protocol

The instructions format is divided into two modes: If $D/\overline{C}$ is set LOW, the current byte is interpreted as a command byte. The command for the module can be found in the datasheet [19]. If $D/\overline{C}$ is set HIGH, the following bytes are stored in the display data RAM. After every data byte, the address counter is incremented automatically. The figure below shows one possible command stream, used to set up the LCD driver.

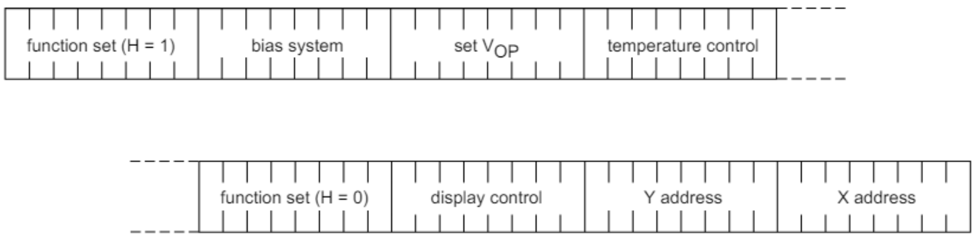


Figure 2.22 Example of command stream to set up the LCD

2.4.3 SPI framework

The SPI subsystem framework in Linux is demonstrated in the below figure.

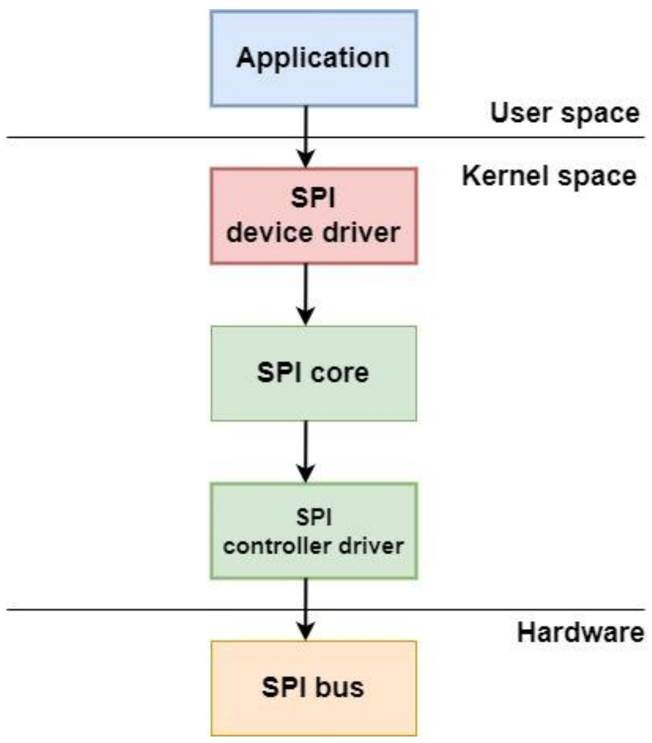


Figure 2.23 Framework of SPI subsystem in Linux

There are 3 main components of the I2C driver framework in the kernel.

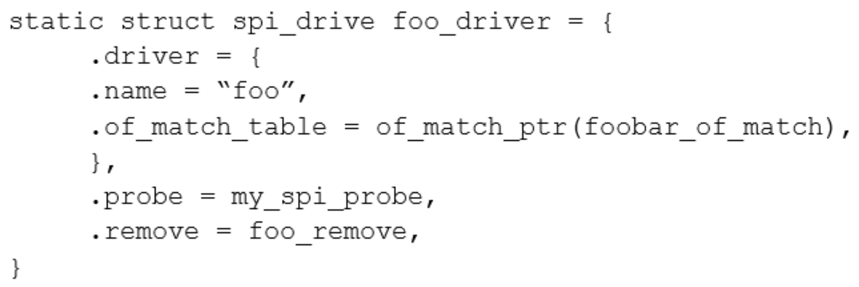
- SPI core: A central component that provides a unified API for both SPI controllers driver and SPI device drivers. It manages the registration and unregistration of SPI bus and SPI devices. It provides functions for transferring data between the SPI master, which is the SPI bus, and the SPI slave, which is the device.
- SPI controller driver: A driver manages a specific SPI bus because a bus is also a device. It abstracts the hardware-specific details of the SPI controller. The SPI controller driver handles the initialization and communication with the SPI controller hardware. It implements functions to send and receive data on the SPI bus. This driver usually written by the manufacturer of the microcontroller.
- SPI device driver: This is a component that most of device driver developers need to pay attention to. The driver manages a specific SPI device called a slave connected to the SPI bus. It abstracts the details of the specific SPI device, providing device-specific functionality to the higher layers of the kernel.

The SPI driver provides a lot of data transferring functions, one of them are `spi_write` function and `spi_read` function, which is has the following fields:

- *spi*: `struct spi_device` of the SPI client;
- *buf*: data to be transfer;

- *len*: size of the data.

The SPI also shares a lot of similarities with the platform device driver like I2C and Serdev driver. The below picture shows an example of the `struct spi_driver`.



```
static struct spi_drive foo_driver = {
     .driver = {
     .name = “foo”,
     .of_match_table = of_match_ptr(foobar_of_match),
     },
     .probe = my_spi_probe,
     .remove = foo_remove,
}
```

Figure 2.24 Example of struct spi_driver

In the next section, we will apply the character device driver skeleton provided at the end of the Chapter 2, LCD NOKIA 5110’s hardware datasheet and the SPI Linux drivers facilities to write a device driver for the LCD NOKIA 5110.

2.4.4 Implementation outline of the LCD NOKIA 5110

2.4.4.4. Driver requirements

For the case of the driver for the LCD NOKIA 5110, we have the following requirements:

- It can write characters and print bitmap image;
- It places the cursor to go to the next line and any position based on the user’s requirement;
- It can clear the screen.

2.4.4.5. Implementation outline

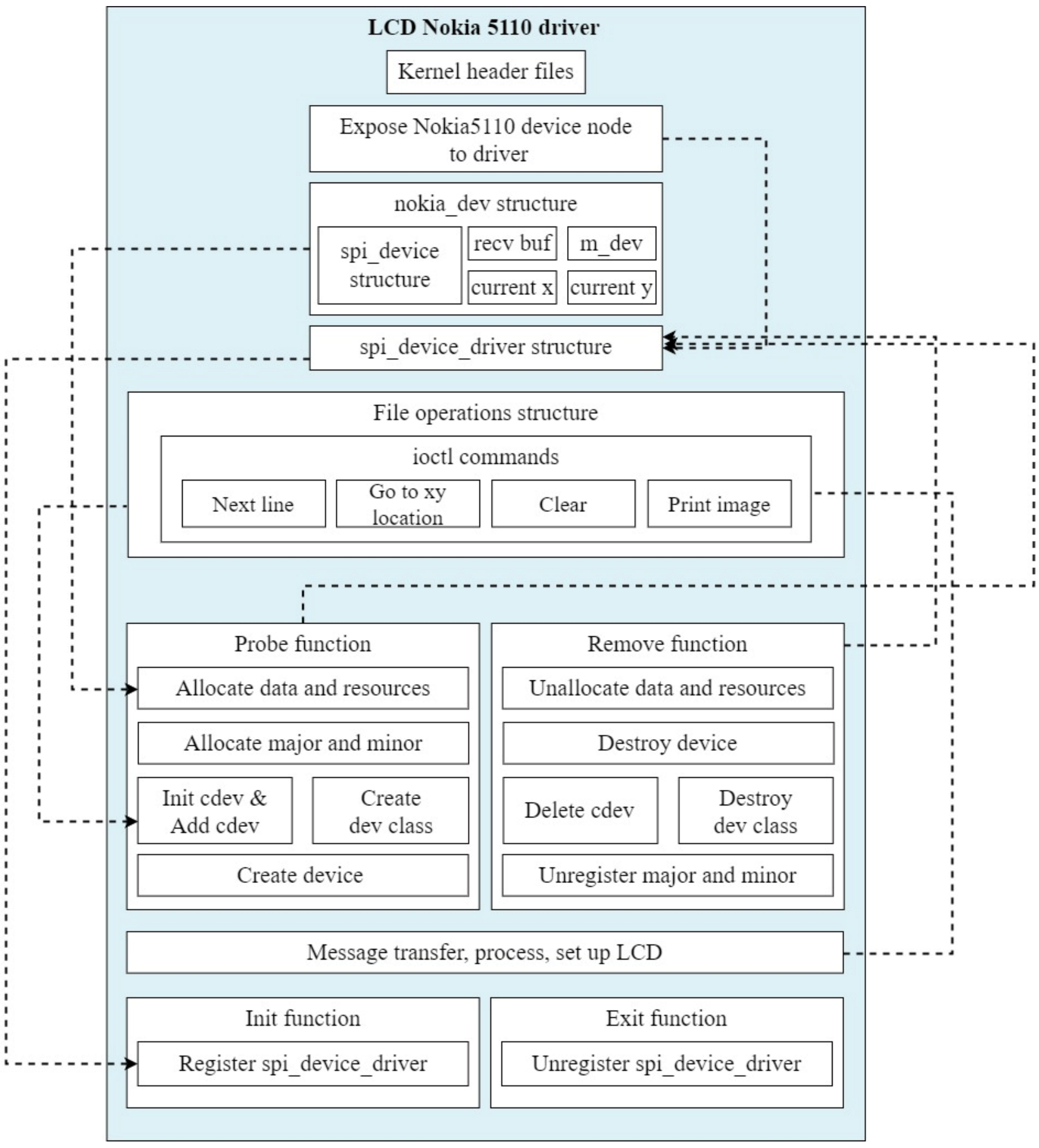


Figure 2.25 LCD Nokia 5110 driver outline

Figure 2.25 is an overview of many components in the LCD Nokia 5110 driver. In this section, we will take a detailed look on these components. The steps to implement them are shown as follows.

- To connect the module LCD Nokia 5110 with the Beagle Bone Black Board, we connect pins as the following table.

Table 2.8. Pin connection of LCD Nokia 5110

LCD Nokia 5110	**Beagle Bone Black**
VCC	3.3V

GND	Ground
RST	P9_11
CE	P9_27
D/C	P9_13
DIN	P9_30
CLK	P9_31
LED	GND

- Now we need to allocate resources for the module by creating a device node in the file *am335x-boneblack.dts*. The details of the device node in the Device Tree can be found in Appendix A.11. section.
- Next, we create the driver code file and declare a `struct nokia5110` which contains `struct spi_device` and some other important members like receive buffer, current X position, current Y position, etc. We want to embed those properties inside the `struct nokia5110` so that this will behave like object in OOP.
- We will declare `probe`, `remove`, `init`, `exit` functions, a `struct file_operations`, and other functions needed to perform desired tasks.
- In order to expose the device to the driver, before calling the `MODULE_DEVICE_TABLE` macro, we must create a `struct of_device_id` that list our devices' names, but in this case we only have one device so there is just one. Details of the `of_device_id` code can be found in the Appendix A.10. section.
- Inside the `probe` function, an instance of the `struct nokia5110` should be allocated on the heap by using `kzalloc()` function, which is the same as `malloc()` in the user space, we need to remember to free it when we unload our device. We now create a character device file to interact with user by allocating the major, minor numbers, create class, register device file with the file operations structure. Then, we must setup the screen by sending setup commands provided in the data sheets and request for some GPIO pins if needed. The driver's data is set using the `spi_set_drvdata()` function. Details of the `probe` function can be found in the Appendix A.10. section.

- The `remove` function should unallocated what is allocated by the `probe` function earlier like class, major, minor numbers, memory allocated for `struct nokia5110`.
- When `probe` and `remove` function is ready, they should be linked to the `struct spi_driver` along with some information about the driver and the `of_match_table` that is defined earlier.
- The next step is to implement the file operations for handling system calls from the user space like `open`, `release`, `read` and `write`. The module LCD Nokia 5110 has no need for `read` system call since its main role is to display output from the user to the screen. And the `write` function is enough to print text to the screen. The building block for all the functions to display data on the screen is the function `spi_write`, which will send 1 byte to the screen. By using this function combined with some algorithms, we can display text and bitmaps like we want. Finally the `ioctl` function is needed to extend the capability of the `write` function. Functions we want to create are placing the cursor to the next line or everywhere we want, clearing the screen and displaying bitmap. To copy the data send from the user to the kernel space and pass to display functions we use the function `copy_from_user`.
- After, we implement the `init` and `exit` functions. In this function we just need to register and unregister the `struct spi_driver`. We just need to pass the `struct spi_driver` instance that was initialized earlier to the `spi_register_driver()`in order to register the driver with the kernel. For the `exit` function we will do the reserve of what that `init` has done, the same as what we do with the `remove` function. Once the `init` and `exit` functions are finished, we assign them with the `module_init()` and `module_exit()` macros to give the driver its entry and exit points.
- Finally, we will write the Makefile script and *make* the driver. Details of the Makefile script is in the Appendix A.12. section.

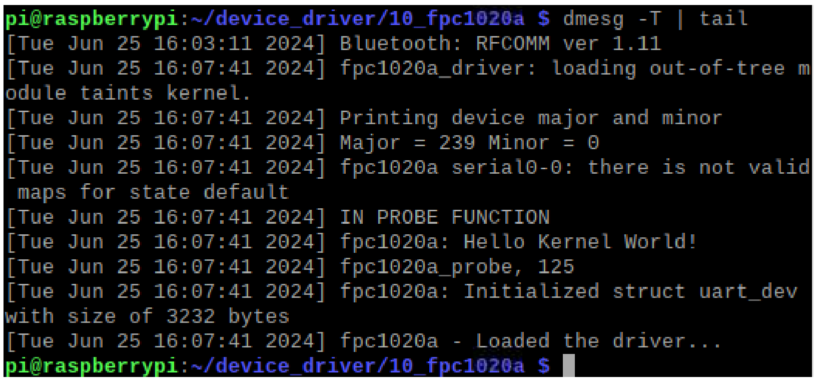
To summarize this chapter, we have utilized device driver basis, character device drivers skeletons and platform device drivers skeletons, along with detailed configurations, implementation steps, and some frameworks for serial, SPI and I2C devices to create different outlines for developing different device drivers. Overall, these drivers’ skeletons have adapted exactly to what we mentioned in the first place.

CHAPTER 3. IMPLEMENTATION RESULTS

From the previous chapter, although the detailed implementation process for each device's driver is different based on each device's function and hardware configurations, the main steps required to deploy each driver are the same. The results after deploying on each device are shown in this chapter.

3.1 Implementation results for FPC1020A

After loading the module and check the kernel message log using *dmesg -T | tail* command, the result of module load is as follows.

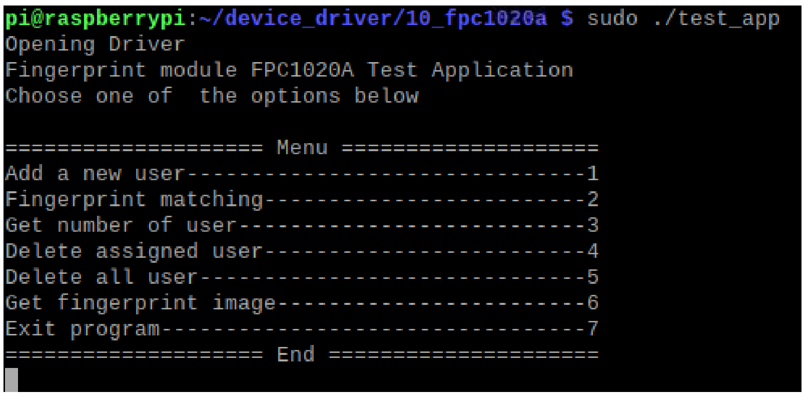


```
pi@raspberrypi:~/device_driver/10_fpc1020a $ dmesg -T | tail
[Tue Jun 25 16:03:11 2024] Bluetooth: RFCOMM ver 1.11
[Tue Jun 25 16:07:41 2024] fpc1020a_driver: loading out-of-tree m
odule taints kernel.
[Tue Jun 25 16:07:41 2024] Printing device major and minor
[Tue Jun 25 16:07:41 2024] Major = 239 Minor = 0
[Tue Jun 25 16:07:41 2024] fpc1020a serial0-0: there is not valid
 maps for state default
[Tue Jun 25 16:07:41 2024] IN PROBE FUNCTION
[Tue Jun 25 16:07:41 2024] fpc1020a: Hello Kernel World!
[Tue Jun 25 16:07:41 2024] fpc1020a_probe, 125
[Tue Jun 25 16:07:41 2024] fpc1020a: Initialized struct uart_dev
with size of 3232 bytes
[Tue Jun 25 16:07:41 2024] fpc1020a - Loaded the driver...
pi@raspberrypi:~/device_driver/10_fpc1020a $
```

Figure 3.1 Message log after loading FPC1020A driver module

Figure 3.1 shows that the module is loaded successfully, with the device's major number is 239 and minor number is 0. The `struct uart_dev` is also successfully initialized with the size of 3232 bytes.

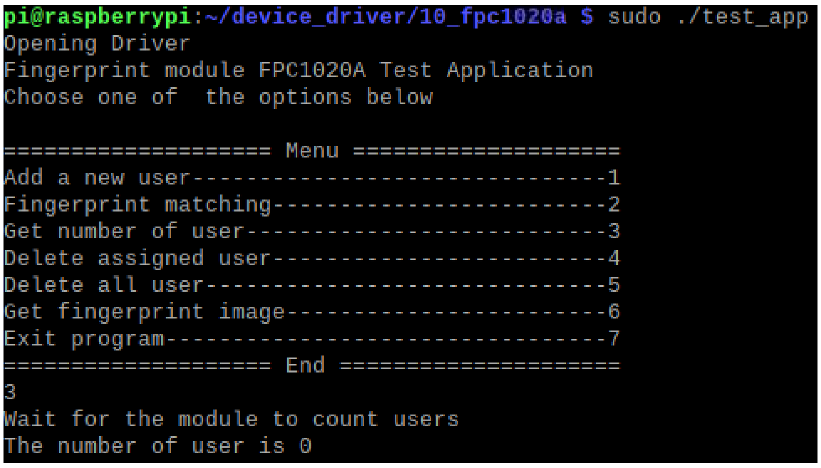
Next, we will test the functionalities of the module with a test user space application. After running the program with *sudo ./test_app* command, the resulting interface is shown as follows.



```
pi@raspberrypi:~/device_driver/10_fpc1020a $ sudo ./test_app
Opening Driver
Fingerprint module FPC1020A Test Application
Choose one of  the options below

=================== Menu ===================
Add a new user------------------------------1
Fingerprint matching------------------------2
Get number of user--------------------------3
Delete assigned user------------------------4
Delete all user-----------------------------5
Get fingerprint image-----------------------6
Exit program--------------------------------7
=================== End ====================
```

Figure 3.2 Userspace test application interface

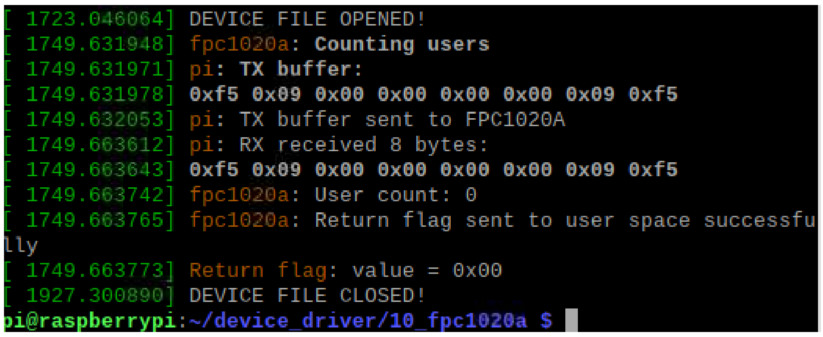
As shown in Figure 3.2, in this program, we have six options to choose from (the last option is just a user space process while the others are commands performed by the kernel), so we will test every case of them and use the dmesg command to check if the timing and data are handled correctly by the kernel or not.



```
pi@raspberrypi:~/device_driver/10_fpc1020a $ sudo ./test_app
Opening Driver
Fingerprint module FPC1020A Test Application
Choose one of  the options below

==================== Menu ====================
Add a new user--------------------------------1
Fingerprint matching-------------------------2
Get number of user---------------------------3
Delete assigned user-------------------------4
Delete all user------------------------------5
Get fingerprint image------------------------6
Exit program---------------------------------7
==================== End =====================
3
Wait for the module to count users
The number of user is 0
```

Figure 3.3 User count before adding any user test case

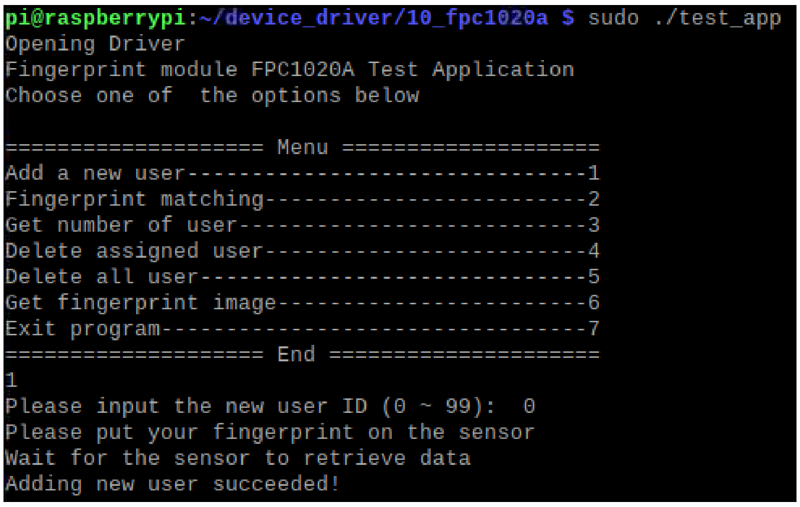
As showcased in Figure 3.3, before adding any user, there is no fingerprint stored in the module as the result returned is 0.



```
[ 1723.046064] DEVICE FILE OPENED!
[ 1749.631948] fpc1020a: Counting users
[ 1749.631971] pi: TX buffer:
[ 1749.631978] 0xf5 0x09 0x00 0x00 0x00 0x00 0x09 0xf5
[ 1749.632053] pi: TX buffer sent to FPC1020A
[ 1749.663612] pi: RX received 8 bytes:
[ 1749.663643] 0xf5 0x09 0x00 0x00 0x00 0x00 0x09 0xf5
[ 1749.663742] fpc1020a: User count: 0
[ 1749.663765] fpc1020a: Return flag sent to user space successfu
lly
[ 1749.663773] Return flag: value = 0x00
[ 1927.300890] DEVICE FILE CLOSED!
pi@raspberrypi:~/device_driver/10_fpc1020a $
```

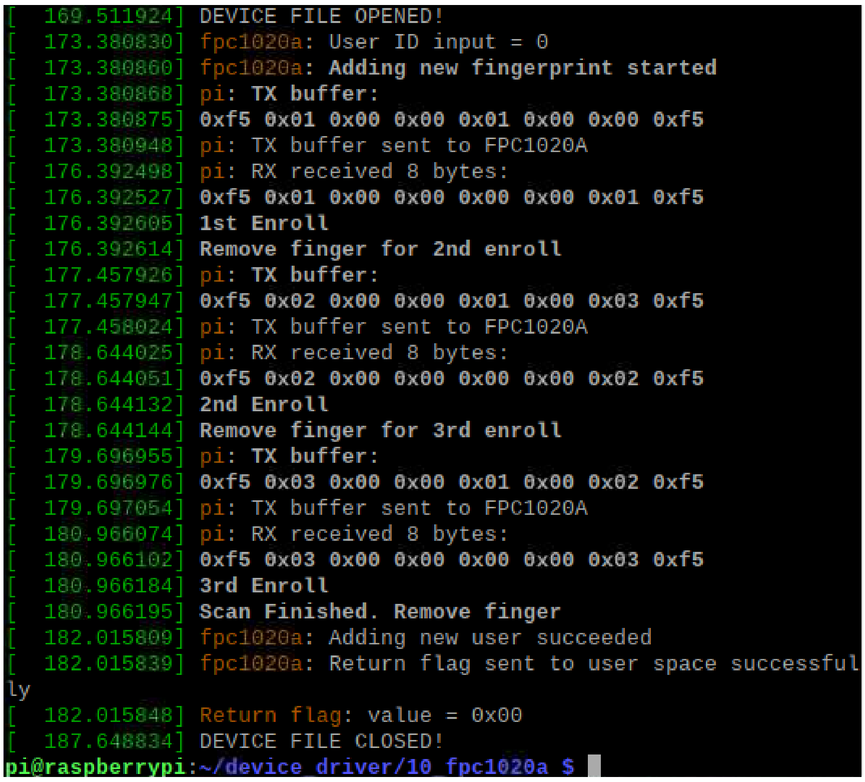
Figure 3.4 User count before adding any user message log

Figure 3.4 also shows that the data sent and received is correct to this case. In addition, the wait time from the point data being sent until the data being received is under 0.032 seconds.



```
pi@raspberrypi:~/device_driver/10_fpc1020a $ sudo ./test_app
Opening Driver
Fingerprint module FPC1020A Test Application
Choose one of  the options below

=================== Menu ===================
Add a new user------------------------------1
Fingerprint matching------------------------2
Get number of user--------------------------3
Delete assigned user------------------------4
Delete all user-----------------------------5
Get fingerprint image-----------------------6
Exit program--------------------------------7
=================== End ====================
1
Please input the new user ID (0 ~ 99):  0
Please put your fingerprint on the sensor
Wait for the sensor to retrieve data
Adding new user succeeded!
```

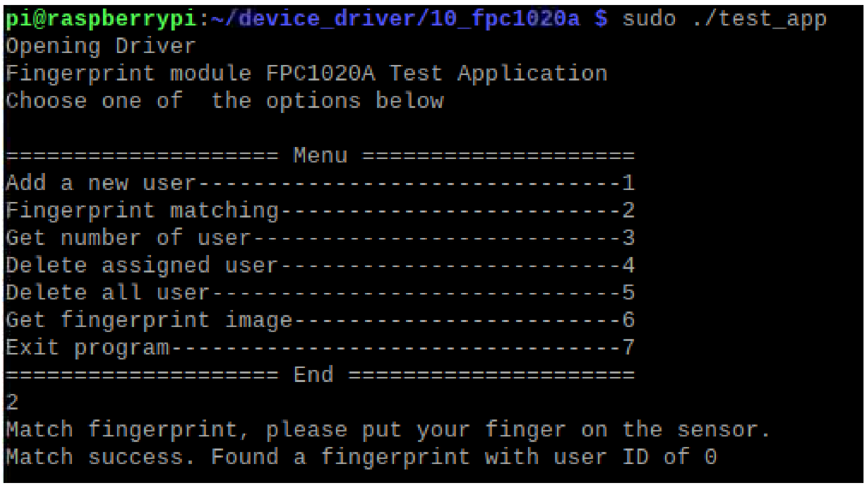
Figure 3.5 Add new user test case



```
[  169.511924] DEVICE FILE OPENED!
[  173.380830] fpc1020a: User ID input = 0
[  173.380860] fpc1020a: Adding new fingerprint started
[  173.380868] pi: TX buffer:
[  173.380875] 0xf5 0x01 0x00 0x00 0x01 0x00 0x00 0xf5
[  173.380948] pi: TX buffer sent to FPC1020A
[  176.392498] pi: RX received 8 bytes:
[  176.392527] 0xf5 0x01 0x00 0x00 0x00 0x00 0x01 0xf5
[  176.392605] 1st Enroll
[  176.392614] Remove finger for 2nd enroll
[  177.457926] pi: TX buffer:
[  177.457947] 0xf5 0x02 0x00 0x00 0x01 0x00 0x03 0xf5
[  177.458024] pi: TX buffer sent to FPC1020A
[  178.644025] pi: RX received 8 bytes:
[  178.644051] 0xf5 0x02 0x00 0x00 0x00 0x00 0x02 0xf5
[  178.644132] 2nd Enroll
[  178.644144] Remove finger for 3rd enroll
[  179.696955] pi: TX buffer:
[  179.696976] 0xf5 0x03 0x00 0x00 0x01 0x00 0x02 0xf5
[  179.697054] pi: TX buffer sent to FPC1020A
[  180.966074] pi: RX received 8 bytes:
[  180.966102] 0xf5 0x03 0x00 0x00 0x00 0x00 0x03 0xf5
[  180.966184] 3rd Enroll
[  180.966195] Scan Finished. Remove finger
[  182.015809] fpc1020a: Adding new user succeeded
[  182.015839] fpc1020a: Return flag sent to user space successful
ly
[  182.015848] Return flag: value = 0x00
[  187.648834] DEVICE FILE CLOSED!
pi@raspberrypi:~/device_driver/10_fpc1020a $
```

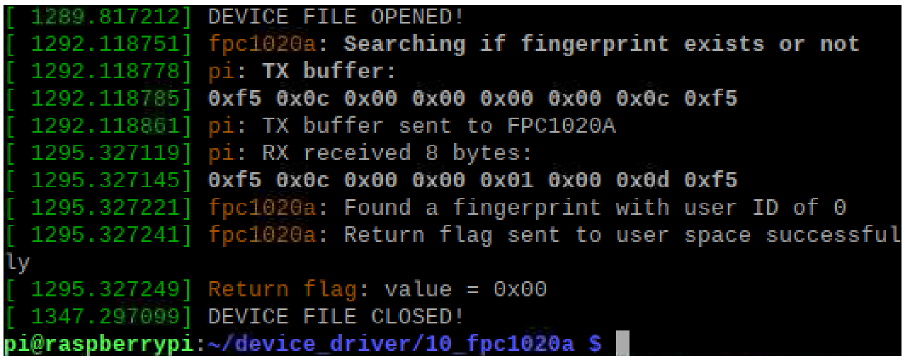
Figure 3.6 Add new user message log

When we choose the add new user option and input the user ID, we will put in our finger and wait for the result and it returned successfully added a new user, as showcased in Figure 3.6. When we check the kernel *dmesg*, the data are all correct, and the wait time for each enrollment are around 0.01, 0.2, and 1.3 seconds respectively, which are acceptable.



```
pi@raspberrypi:~/device_driver/10_fpc1020a $ sudo ./test_app
Opening Driver
Fingerprint module FPC1020A Test Application
Choose one of  the options below

==================== Menu ====================
Add a new user-----------------------------1
Fingerprint matching-----------------------2
Get number of user-------------------------3
Delete assigned user-----------------------4
Delete all user----------------------------5
Get fingerprint image----------------------6
Exit program-------------------------------7
==================== End =====================
2
Match fingerprint, please put your finger on the sensor.
Match success. Found a fingerprint with user ID of 0
```

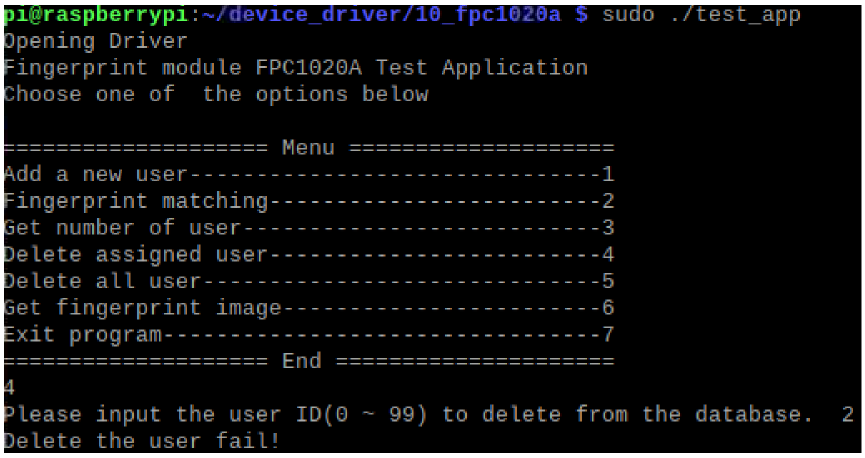
Figure 3.7 Fingerprint matching test case



```
[ 1289.817212] DEVICE FILE OPENED!
[ 1292.118751] fpc1020a: Searching if fingerprint exists or not
[ 1292.118778] pi: TX buffer:
[ 1292.118785] 0xf5 0x0c 0x00 0x00 0x00 0x00 0x0c 0xf5
[ 1292.118861] pi: TX buffer sent to FPC1020A
[ 1295.327119] pi: RX received 8 bytes:
[ 1295.327145] 0xf5 0x0c 0x00 0x00 0x01 0x00 0x0d 0xf5
[ 1295.327221] fpc1020a: Found a fingerprint with user ID of 0
[ 1295.327241] fpc1020a: Return flag sent to user space successful
ly
[ 1295.327249] Return flag: value = 0x00
[ 1347.297099] DEVICE FILE CLOSED!
pi@raspberrypi:~/device_driver/10_fpc1020a $
```

Figure 3.8 Fingerprint matching message log

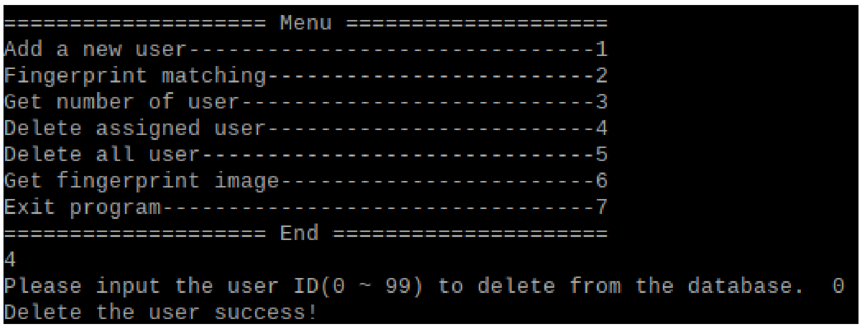
When we perform the fingerprint matching option with the same finger that we performed inserting in the previous, the result returned 0 which is correct, as showcased in both Figure 3.7 and 3.8.



```
pi@raspberrypi:~/device_driver/10_fpc1020a $ sudo ./test_app
Opening Driver
Fingerprint module FPC1020A Test Application
Choose one of  the options below

==================== Menu ====================
Add a new user-----------------------------1
Fingerprint matching-----------------------2
Get number of user-------------------------3
Delete assigned user-----------------------4
Delete all user----------------------------5
Get fingerprint image----------------------6
Exit program-------------------------------7
==================== End =====================
4
Please input the user ID(0 ~ 99) to delete from the database.  2
Delete the user fail!
```

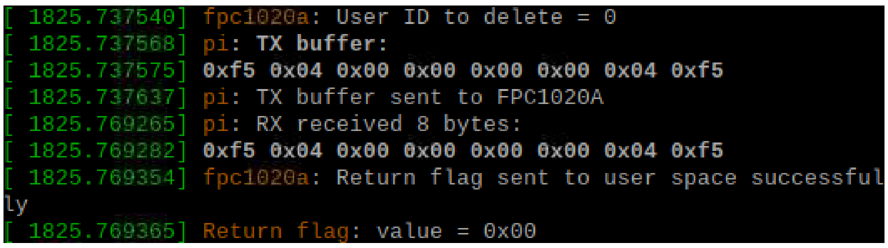
Figure 3.9 Delete user with ID of 2 test case

When we delete the user with the ID that does not exist in the module, like in Figure 3.9, the result is fail.



```
==================== Menu ===================
Add a new user-----------------------------1
Fingerprint matching-----------------------2
Get number of user-------------------------3
Delete assigned user-----------------------4
Delete all user----------------------------5
Get fingerprint image----------------------6
Exit program-------------------------------7
==================== End ====================
4
Please input the user ID(0 ~ 99) to delete from the database.  0
Delete the user success!
```

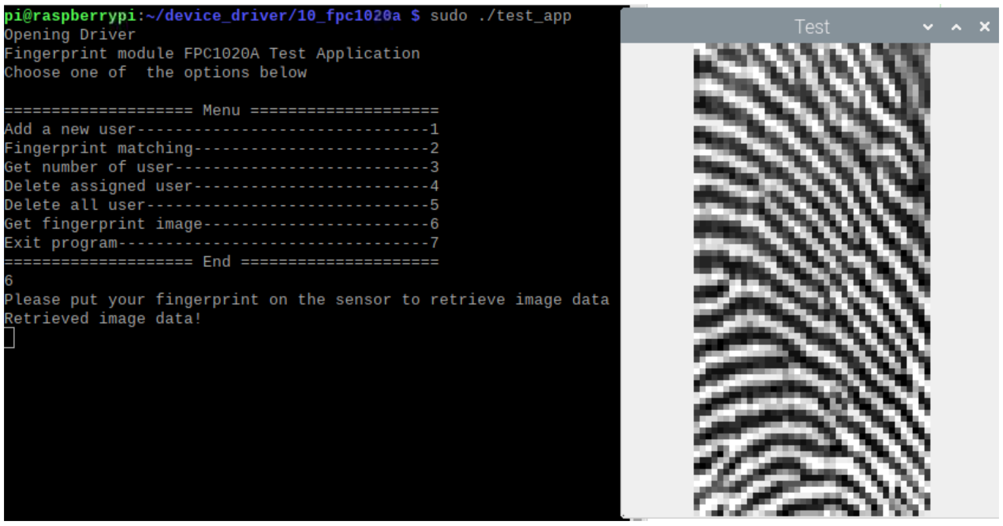
Figure 3.10 Delete user with ID of 0 test case



```
[ 1825.737540] fpc1020a: User ID to delete = 0
[ 1825.737568] pi: TX buffer:
[ 1825.737575] 0xf5 0x04 0x00 0x00 0x00 0x00 0x04 0xf5
[ 1825.737637] pi: TX buffer sent to FPC1020A
[ 1825.769265] pi: RX received 8 bytes:
[ 1825.769282] 0xf5 0x04 0x00 0x00 0x00 0x00 0x04 0xf5
[ 1825.769354] fpc1020a: Return flag sent to user space successful
ly
[ 1825.769365] Return flag: value = 0x00
```

Figure 3.11 Delete user with ID of 0 message log

If we delete the user with ID of 0 instead, the result would be true, as shown in Figure 3.10 and 3.11. The result would be the same if we perform the delete all user option.



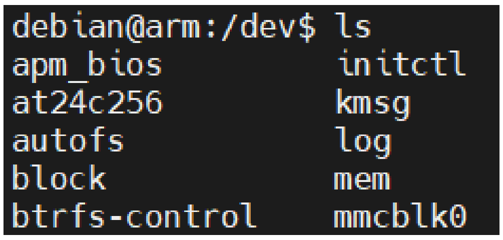
```
pi@raspberrypi:~/device_driver/10_fpc1020a $ sudo ./test_app
Opening Driver
Fingerprint module FPC1020A Test Application
Choose one of  the options below

=================== Menu ===================
Add a new user-----------------------------1
Fingerprint matching-----------------------2
Get number of user-------------------------3
Delete assigned user-----------------------4
Delete all user----------------------------5
Get fingerprint image----------------------6
Exit program-------------------------------7
=================== End ====================
6
Please put your fingerprint on the sensor to retrieve image data
Retrieved image data!
```

Figure 3.12 Get fingerprint image test case

We tested the last option of getting fingerprint image and the result returned, as shown in Figure 3.12, is a 40 x 80 grayscale picture that is identical to the finger that we input.

3.2 Implementation results for EEPROM AT24C256

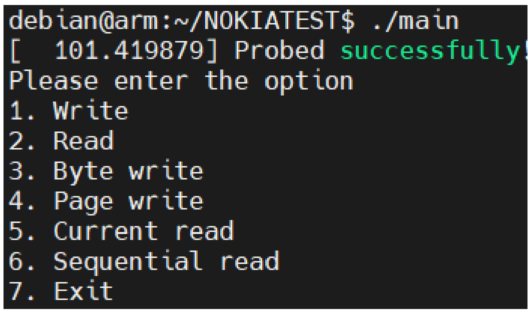
After loading the module, we can check the *dev/* directory whether the module has been loaded successfully or not. The result is shown in the figure below.



```
debian@arm:/dev$ ls
apm_bios          initctl
at24c256          kmsg
autofs            log
block             mem
btrfs-control     mmcblk0
```

Figure 3.13 at24c256 device file

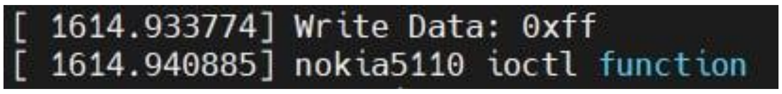
The Figure 3.13 shows the instance of the device file of AT24C256 has shown up in the */dev* directory, which means the module has been loaded successfully. We now run the testing application by using command *./main.* The following picture illustrates the interface with the user.



```
debian@arm:~/NOKIATEST$ ./main
[  101.419879] Probed successfully
Please enter the option
1. Write
2. Read
3. Byte write
4. Page write
5. Current read
6. Sequential read
7. Exit
```

Figure 3.14 AT24C256 EEPROM testing interface

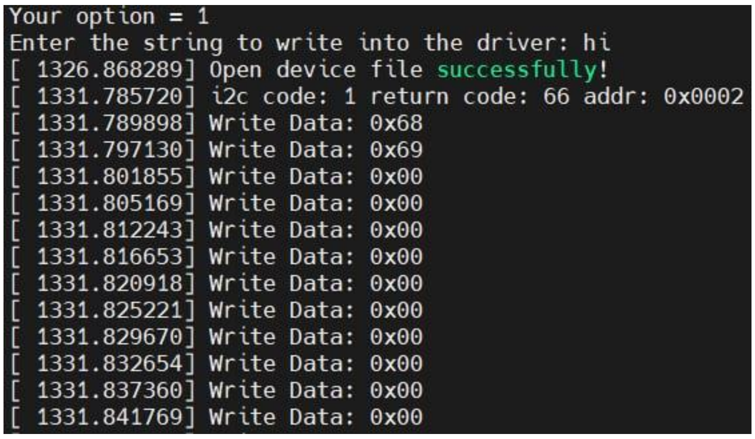
The probe message shows up when we load the module, in the Figure 3.14 above it means that the `probe` has been successfully called. We now try the writing one byte function and the result is shown in the below figure.



```
[ 1614.933774] Write Data: 0xff
[ 1614.940885] nokia5110 ioctl function
```

Figure 3.15 Write one byte function

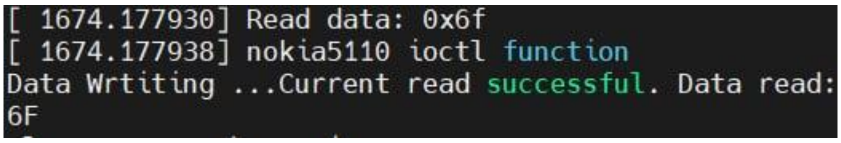
The function has been successfully written to the EEPROM as depicted in the Figure 3.15. Next, we try the function to write one page of 64 bytes and by typing text which shown as follows.



```
Your option = 1
Enter the string to write into the driver: hi
[ 1326.868289] Open device file successfully!
[ 1331.785720] i2c code: 1 return code: 66 addr: 0x0002
[ 1331.789898] Write Data: 0x68
[ 1331.797130] Write Data: 0x69
[ 1331.801855] Write Data: 0x00
[ 1331.805169] Write Data: 0x00
[ 1331.812243] Write Data: 0x00
[ 1331.816653] Write Data: 0x00
[ 1331.820918] Write Data: 0x00
[ 1331.825221] Write Data: 0x00
[ 1331.829670] Write Data: 0x00
[ 1331.832654] Write Data: 0x00
[ 1331.837360] Write Data: 0x00
[ 1331.841769] Write Data: 0x00
```

Figure 3.16 Write one page function

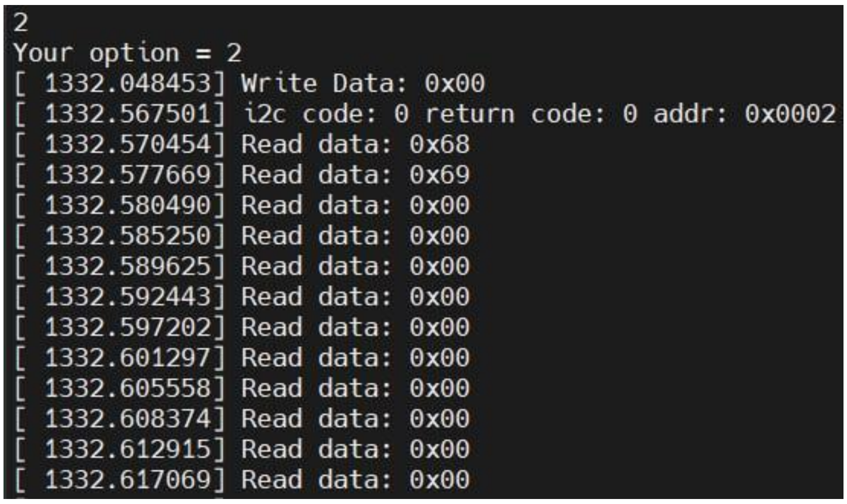
The Figure 3.16 shows the message write to the EEPROM and the return code means that 66 bytes has been written, the additional 2 bytes is for the word addresses we want to write to. We then test the reading one byte function, which is illustrated in the following figure.



```
[ 1674.177930] Read data: 0x6f
[ 1674.177938] nokia5110 ioctl function
Data Wrtiting ...Current read successful. Data read:
6F
```

Figure 3.17 Read one byte function

A message read from the read one byte function is the same as the first byte from the write page function in the Figure 3.17. Finally, we test the read a page function and get the result, which is the same as the data written by the writing one page function, shown in the following picture.



```
2
Your option = 2
[ 1332.048453] Write Data: 0x00
[ 1332.567501] i2c code: 0 return code: 0 addr: 0x0002
[ 1332.570454] Read data: 0x68
[ 1332.577669] Read data: 0x69
[ 1332.580490] Read data: 0x00
[ 1332.585250] Read data: 0x00
[ 1332.589625] Read data: 0x00
[ 1332.592443] Read data: 0x00
[ 1332.597202] Read data: 0x00
[ 1332.601297] Read data: 0x00
[ 1332.605558] Read data: 0x00
[ 1332.608374] Read data: 0x00
[ 1332.612915] Read data: 0x00
[ 1332.617069] Read data: 0x00
```

Figure 3.18 Reading one page function

3.3 Implementation results for LCD NOKIA 5110

After loading the module, we get the following notification which is shown in the figure below.

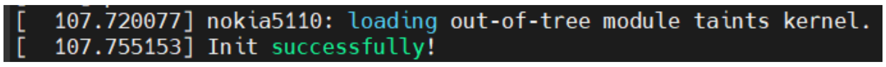

```
[  107.720077] nokia5110: loading out-of-tree module taints kernel.
[  107.755153] Init successfully!
```

Figure 3.19 Messages after loading module LCD Nokia 5110 driver

The Figure 3.19 means that we have successfully load the module into the kernel. The module shows the *init* message which is illustrated in the below figure.

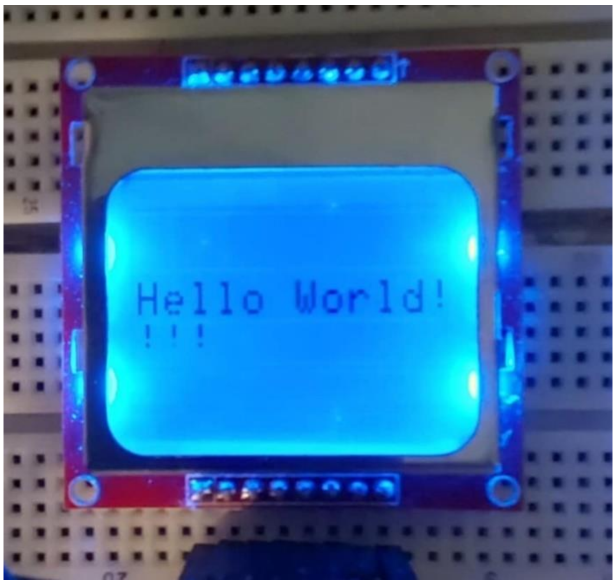


Figure 3.20 Init message after loading the module

Next, we test the functionalities of the module by using command *./testnokia*, the resulting interface is displayed as follows.

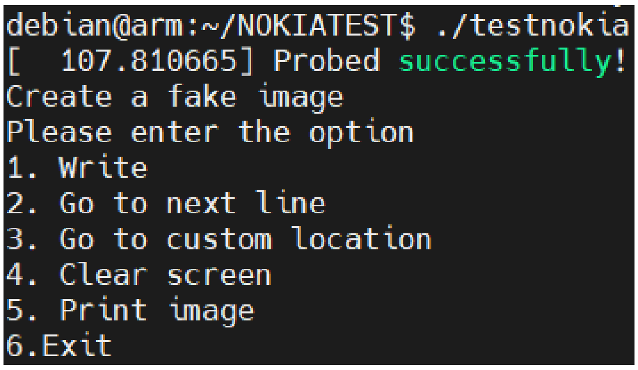

```
debian@arm:~/NOKIATEST$ ./testnokia
[  107.810665] Probed successfully!
Create a fake image
Please enter the option
1. Write
2. Go to next line
3. Go to custom location
4. Clear screen
5. Print image
6.Exit
```

Figure 3.21 Test application interface

The Figure 3.21 shows the options that can be chosen by the user. Somehow, the *probe* message shows up after we run the application, it shows that the `probe` function has been called successfully. We now clear the screen by choosing option 4, the screen is displayed as follows.

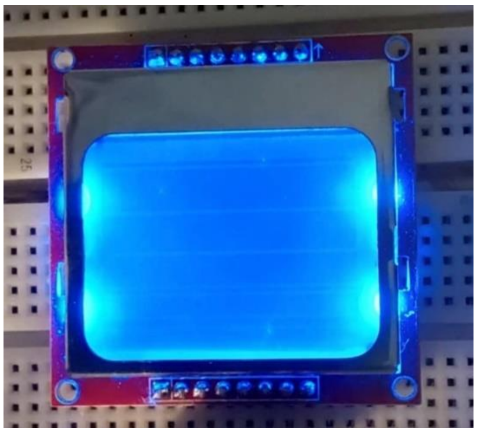


Figure 3.22 Screen after clearing

Next we write text in the first line using the option 3 and option 1 and the result is shown in the following figure.

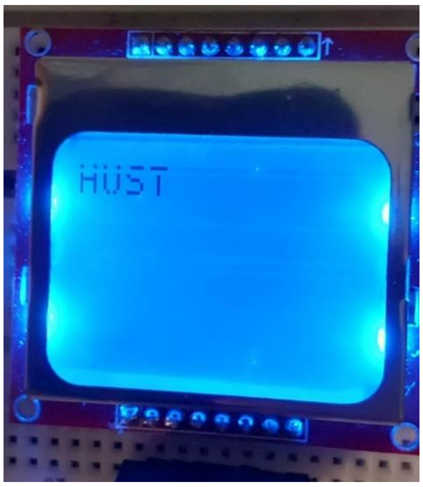


Figure 3.23 Writing to the screen

Finally, we test the printing bitmap function by choosing option 5 and the result is as follows.

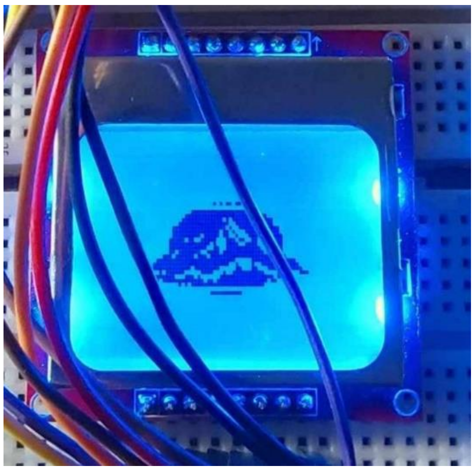


Figure 3.24 Printing bitmap image

The test results clearly demonstrate that all three devices functioned correctly, meeting the technical specifications and operational criteria set forth.

CONCLUSION

Conclusion

After evaluating the results and functional effectiveness of the developed drivers, we have come to the conclusion that the proposed outline and the method of development are solid and highly applicable in many other devices. The completion level of the drivers is high, ready for use by other programs.

Moreover, though the project may seem small and simple, but it could be a good source of reference especially for Linux kernel fresher developers to start working on this platform with an ease.

By doing this project, we, from the point of view of ones who are new to this topic, have acquired some basic knowledge and skills needed in a junior kernel developer, have gained more experience with working with many hardware, and above all, have acquired even more experience working as a team and communicate.

Future improvements and works

In the framework of this project, although we have made it able to make the basic functionalities of these devices work, there is still room for improvements in handling errors and better performance time. In addition, there are still many lower level layers with more sophisticated structures that we could dig in deeper to understand how things work in those platforms.

REFERENCES

[1] J. Madieu, "Introduction to Kernel Development," in *Linux Device Drivers Development*, Packt Publishing Ltd., 2017, p. 8.

[2] J. Corbet, A. Rubini and G. Kroah-Hartman, "An Introduction to Device Drivers," in *Linux Device Drivers, Third Edition*, O'Reilly, 2005, pp. 1-8.

[3] J. Madieu, "Device Driver Basis," in *Linux Device Drivers Development*, Packt Publishing Ltd., 2017, pp. 18-20, 34-42.

[4] Oracle Corporation, "Differences Between Kernel Modules and User Programs," 2010. [Online]. Available: https://docs.oracle.com/cd/E19253-01/817-5789/emjjr/index.html. [Accessed 08 06 2024].

[5] J. Corbet, A. Rubini and G. Kroah-Hartman, "Char Drivers," in *Linux Device Drivers, Third Edition*, O'Relly, 2005, pp. 42-70.

[6] J. Madieu, "Character Device Drivers," in *Linux Device Drivers Development*, Packt Publishing Ltd., 2017, pp. 93-97.

[7] J. Madieu, "Kernel Facilities and Helper Fucntions," in *Linux Device Drivers Development*, Packt Publishing Ltd., 2017, pp. 51-53.

[8] J. Corbet, A. Rubini and G. Kroah-Hartman, "Advanced Char Driver Operations," in *Linux Device Drivers, Third Edition*, O'Reilly Media, 2005, pp. 148-150.

[9] J. Madieu, "Platform Device Drivers," in *Linux Device Drivers Development*, Packt Publishing Ltd., 2017, pp. 116-117, 126.

[10] J. Madieu, "Pin Control and GPIO Subsystem," in *Linux Device Drivers Development*, Packt Publishing, 2017, p. 362.

[11] J. Madieu, "The Concept of a Device Tree," in *Linux Device Drivers Development*, Packt Publishing Ltd., 2017, p. 146.

[12] J. Corbet , A. Rubini and G. Kroah-Hartman, "The Linux Device Model," in *Linux Device Drivers, Third Edition*, O'Reilly, 2005, pp. 363, 370.

[13 J. Madieu, "The Linux Device Model," in *Linux Device Drivers*

] *Development*, Packt Publishing Ltd., 2017, p. 348.

[14] D. P. Bovert and M. Cesati, "I/O Architecture and Device Drivers," in *Understanding the Linux Kernel, Third Edition*, O'Reily, 2005, pp. 532, 533, 534, 535.

[15] Raspberry Pi Ltd., "Configuration - Raspberry Pi Documentation," 2024. [Online]. Available: https://www.raspberrypi.com/documentation/computers/configuration.html#configure-uarts. [Accessed 22 June 2024].

[16] Biovo, "FPC1020A Capacitive Fingerprint Module".

[17] J. Hovold, "The Serial Device Bus," in *Embedded Linux Conference Europe*, 2017.

[18] Atmel, "Microchip," Microchip, [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8568-SEEPROM-AT24C256C-Datasheet.pdf.

[19] P. Semiconductors, "Sparkfun," Philips Semiconductors, 1999. [Online]. Available: https://www.sparkfun.com/datasheets/LCD/Monochrome/Nokia5110.pdf.

APPENDIX

A1. Details of FPC1020A driver's header file

```
#ifndef FPC1020A_DRIVER_h
#define FPC1020A_DRIVER_h

// Defining FPC1020A module's commands and flags
#define ACK_SUCCESS             0x00  // Operation succeeded
#define ACK_FAIL                0x01  // Operation failed
#define ACK_FULL                0x04  // Fingerprint database is full
#define ACK_NO_USER             0x05  // Users do not exist
#define ACK_USER_OCCUPIED       0x06  // User ID already exists
#define ACK_USER_EXIST          0x07  // Fingerprint already exists
#define ACK_TIMEOUT             0x08  // Acquisition timeout

#define CMD_ENROLL1             0x01  // Add new fingerprint 1st time
#define CMD_ENROLL2             0x02  // Add new fingerprint 2nd time
#define CMD_ENROLL3             0x03  // Add new fingerprint 3rd time
#define CMD_DELETE_USER         0x04  // Delete assigned user
#define CMD_DELETE_ALL          0x05  // Delete all users
#define CMD_USER_COUNT          0x09  // Get the number of users
#define CMD_IDENTIFY            0x0B  // Fingerprint matching 1:1
#define CMD_SEARCH              0x0C  // Fingerprint matching 1:N
#define CMD_GET_USER_ID         0x2B  // Get user ID and user
permission
#define CMD_GET_IMAGE           0x24  // Get user's fingerprint image

#define DATA_START              0xF5  // Data start byte
#define DATA_END                0xF5  // Data end byte

#define IRG_NO 11

// Defining ioctl commands to be called
#define ADD_NEW_USER            _IOW(0xF5, 1, uint8_t*)
#define FP_SEARCH               _IOR(0xF5, 2, uint8_t*)
#define USER_COUNT              _IOR(0xF5, 3, uint8_t*)
```

```
#define DELETE_A_USER         _IOW(0xF5, 4, uint8_t*)
#define DELETE_ALL_USER       _IO(0xF5, 5)
#define GET_FP_IMAGE          _IOR(0xF5, 6, uint8_t*)
#define GET_RT_FLAG           _IOR(0xF5, 10, unsigned char*)

#endif
```

A2. Details of FPC1020A driver's source code

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/serdev.h>
#include <linux/types.h>
#include <linux/mod_devicetable.h>
#include <linux/property.h>
#include <linux/platform_device.h>
#include <linux/of.h>
#include <linux/err.h>
#include <linux/fs.h>
#include <linux/slab.h>
#include <linux/ioctl.h>
#include <linux/uaccess.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/kdev_t.h>
#include <linux/string.h>
#include <linux/time.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/wait.h>

#include "fpc1020a_driver.h"

DECLARE_WAIT_QUEUE_HEAD(wait_queue_tx);
```

```
/* Variables for module init */
dev_t dev = 0;
static struct class *dev_class;
static struct cdev fpc1020a_cdev;
int temp_flag = 0;

unsigned short TX_BUF_SIZE = 8;
unsigned short i = 0, j = 0, pos = 0;

// FPC1020A device struct
struct uart_dev {
	struct serdev_device *serdev;
	uint8_t *tx_buf; //fpc1020a tx buf is pi rx buf
	uint8_t *rx_buf; //fpc1020a rx buf is pi tx buf
	unsigned int tx_count;
	char tx_end_flag;
	unsigned int u_id;
	unsigned int u_count;
	unsigned char rtflag;
	uint8_t img_matrix[80][40];
};
struct uart_dev *fpc1020a = NULL;

/* Declare the probe and remove functions */
static int fpc1020a_probe(struct serdev_device *serdev);
static void fpc1020a_remove(struct serdev_device *serdev);

/* Init + Exit + File ops function prototypes */
static int __init fpc1020a_driver_init(void);
static void __exit fpc1020a_driver_exit(void);
static int fpc1020a_open(struct inode *inode, struct file *file);
static int fpc1020a_release(struct inode *inode, struct file *file);
static ssize_t fpc1020a_read(struct file *filp, char __user *buf,
size_t len, loff_t *off);
static ssize_t fpc1020a_write(struct file *filp, const char *buf,
size_t len, loff_t *off);
static long fpc1020a_ioctl(struct file *file, unsigned int cmd,
unsigned long arg);


/* FPC1020A module functionality */
static char fpc1020a_gen_check_sum(unsigned char wLen, unsigned char
*ptr);
static void fpc1020a_send_package(unsigned char wLen, unsigned char
*ptr);
static char fpc1020a_check_package(unsigned char cmd);
static char fpc1020a_search(void);
//static void fpc1020a_identify(unsigned int u_id);
static void fpc1020a_enroll1(unsigned int u_id);
static void fpc1020a_enroll2(unsigned int u_id);
static void fpc1020a_enroll3(unsigned int u_id);
static char fpc1020a_enroll(unsigned int u_id);
static char fpc1020a_delete_all(void);
static char fpc1020a_delete_user(unsigned int u_id);
static char fpc1020a_user_count(void);
//static char fpc1020a_get_user_id(void);
static char fpc1020a_get_image(void);


static struct of_device_id fpc1020a_of_match_ids[] = {
	{
		.compatible = "fpc,fpc1020a",
	}, { /* sentinel */ }
};


MODULE_DEVICE_TABLE(of, fpc1020a_of_match_ids);


/* Callback is called whenever a character is received */
static int uart_receive(struct serdev_device *serdev, const uint8_t
*buff, size_t size)
{
	// Copying received character to fpc1020a TX buffer (pi's RX)
and count
	unsigned int i = 0;
	memcpy(fpc1020a->tx_buf + fpc1020a->tx_count, buff, size);
	fpc1020a->tx_count++;
	if(fpc1020a->tx_count == TX_BUF_SIZE) {
```

```
            printk(KERN_INFO "pi: RX received %d bytes:\n",
fpc1020a->tx_count);
            for(i = 0; i < fpc1020a->tx_count; i++) {
                  printk(KERN_CONT "0x%02x ", *(fpc1020a->tx_buf +
i));
            }
            fpc1020a->tx_end_flag = 1;
            wake_up_interruptible(&wait_queue_tx);
      }
      return size;
}

static const struct serdev_device_ops serdev_ops = {
      .receive_buf = uart_receive,
};

static struct serdev_device_driver fpc1020a_driver = {
      .probe = fpc1020a_probe,
      .remove = fpc1020a_remove,
      .driver = {
            .name = "fpc1020a",
            .owner = THIS_MODULE,
            .of_match_table = fpc1020a_of_match_ids,
      },
};

/* This function is called on loading the driver */
static int fpc1020a_probe(struct serdev_device *serdev)
{
      printk(KERN_INFO "IN PROBE FUNCTION\n");
      printk(KERN_INFO "fpc1020a: Hello Kernel World!\n");
      printk(KERN_INFO "%s, %d\n", __func__, __LINE__);

      // Memory allocating our uart device on the heap
      // NOTE: 'devm_kzalloc' automatically free memory when unload
our device
      fpc1020a = devm_kzalloc(&serdev->dev, sizeof(struct uart_dev),
GFP_KERNEL);
```

```
	if(!fpc1020a){
		printk(KERN_ERR "UART device memory allocation
failed\n");
		return -1;
	}
	printk("fpc1020a: Initialized struct uart_dev with size of %d
bytes\n", sizeof(struct uart_dev));
	fpc1020a->serdev = serdev;
	serdev_device_set_drvdata(serdev, fpc1020a);

	serdev_device_set_client_ops(fpc1020a->serdev, &serdev_ops);
	if(serdev_device_open(fpc1020a->serdev) != 0){
		printk(KERN_ERR "Cannot open serial device\n");
		return -1;
	}
	serdev_device_set_baudrate(fpc1020a->serdev, 19200);
	serdev_device_set_flow_control(fpc1020a->serdev, false);
	serdev_device_set_parity(fpc1020a->serdev,
SERDEV_PARITY_NONE);

	// Buffer memory allocation
	fpc1020a->rx_buf = kzalloc(8 * sizeof(uint8_t), GFP_KERNEL);
	if(fpc1020a->rx_buf == NULL){
		printk(KERN_ERR "kzalloc for UART device RX buffer
failed\n");
		return -1;
	}

	fpc1020a->tx_buf = kzalloc(8 * sizeof(uint8_t), GFP_KERNEL);
	if(fpc1020a->tx_buf == NULL){
		printk(KERN_ERR "kzalloc for UART device TX buffer
failed\n");
		return -1;
	}

	memset(fpc1020a->rx_buf, 0, 8);
	memset(fpc1020a->tx_buf, 0, 8);
```

```
      fpc1020a->tx_count = 0;
      fpc1020a->tx_end_flag = 0; // Not end = 0, end = 1

      return 0;
}

/* This function is called on unloading the driver */
static void fpc1020a_remove(struct serdev_device *serdev)
{
      printk(KERN_INFO "IN REMOVE FUNCTION\n");
      printk(KERN_INFO "%s, %d\n", __func__, __LINE__);
      fpc1020a = serdev_device_get_drvdata(serdev);
      if(!fpc1020a){
            printk("Get private data failure\n");
      } else {
            kfree(fpc1020a->tx_buf);
            kfree(fpc1020a->rx_buf);
            printk(KERN_INFO "fpc1020a: Goodbye Kernel World!\n");
            serdev_device_close(fpc1020a->serdev);
      }
}

/* File operations structure */
static struct file_operations fops = {
      .owner                  = THIS_MODULE,
      .read             = fpc1020a_read,
      .write                  = fpc1020a_write,
      .open             = fpc1020a_open,
      .release          = fpc1020a_release,
      .unlocked_ioctl   = fpc1020a_ioctl,
};

static int fpc1020a_open(struct inode *inode, struct file *file){
      printk(KERN_INFO "DEVICE FILE OPENED!\n");
      return 0;
}
```

```
static int fpc1020a_release(struct inode *inode, struct file *file){
      printk(KERN_INFO "DEVICE FILE CLOSED!\n");
      return 0;
}

static ssize_t fpc1020a_read(struct file *filp, char __user *buf,
size_t len, loff_t *off){
      printk(KERN_INFO "DEVICE FILE READ!\n");
      return 0;
}

static ssize_t fpc1020a_write(struct file *filp, const char *buf,
size_t len, loff_t *off){
      printk(KERN_INFO "DEVICE FILE WRITE!\n");
      return len;
}

static long fpc1020a_ioctl(struct file *file, unsigned int cmd,
unsigned long arg)
{
      switch(cmd) {
            case GET_RT_FLAG:
                  if(copy_to_user((unsigned char*) arg, &(fpc1020a-
>rtflag), sizeof(fpc1020a->rtflag))) {
                        printk(KERN_ERR "fpc1020a: ERROR Couldn't
send return flag to user space\n");
                  }
                  printk(KERN_INFO "fpc1020a: Return flag sent to
user space successfully\n");
                  printk(KERN_INFO "Return flag: value = 0x%02x\n",
fpc1020a->rtflag);
                  break;

            case ADD_NEW_USER:
                  fpc1020a->tx_count = 0;
                  fpc1020a->u_id = 0;
                  if(copy_from_user(&(fpc1020a->u_id), (unsigned
int*) arg, sizeof(fpc1020a->u_id))) {
```

```
                        printk(KERN_ERR "fpc1020a: ERROR Couldn't
get user ID input!\n");
                  }
                  printk(KERN_INFO "fpc1020a: User ID input = %d\n",
fpc1020a->u_id);
                  if(fpc1020a->u_id >= 0 && fpc1020a->u_id <= 99) {
                        fpc1020a_enroll(fpc1020a->u_id);
                  } else {
                        printk("User ID invalid\n");
                  }

                  if(fpc1020a->rtflag == ACK_SUCCESS) {
                        printk(KERN_INFO "fpc1020a: Adding new user
succeeded\n");
                  } else {
                        printk(KERN_ERR "fpc1020a: Adding new user
failed\n");
                  }
                  break;

            case FP_SEARCH:
                  fpc1020a->tx_count = 0;
                  fpc1020a->u_id = 0;
                  printk("fpc1020a: Searching if fingerprint exists
or not\n");
                  fpc1020a_search();
                  if(fpc1020a->rtflag == ACK_SUCCESS) {
                        printk(KERN_INFO "fpc1020a: Found a
fingerprint with user ID of %d\n", fpc1020a->u_id);
                        if(copy_to_user((unsigned int*) arg,
&(fpc1020a->u_id), sizeof(fpc1020a->u_id))) {
                              printk(KERN_ERR "fpc1020a: Couldn't
send ID to user space\n");
                        }
                  } else {
                        printk(KERN_INFO "fpc1020a: Fingerprint does
not exist!\n");
                  }
                  break;
            case USER_COUNT:
                  fpc1020a->tx_count = 0;
                  printk("fpc1020a: Counting users\n");
                  fpc1020a_user_count();
                  if(fpc1020a->rtflag == ACK_SUCCESS) {
                        printk(KERN_INFO "fpc1020a: User count:
%d\n", fpc1020a->u_count);
                        if(copy_to_user((unsigned int*) arg,
&(fpc1020a->u_count), sizeof(fpc1020a->u_count))) {
                              printk(KERN_ERR "fpc1020a: Couldn't
send count to user space\n");
                        }
                  } else {
                        printk(KERN_INFO "fpc1020a: Couldn't
retrieve user count\n");
                  }
                  break;

            case DELETE_A_USER:
                  fpc1020a->tx_count = 0;
                  if(copy_from_user(&(fpc1020a->u_id), (unsigned
int*) arg, sizeof(fpc1020a->u_id))){
                        printk(KERN_ERR "Data write: ERROR!\n");
                  }
                  printk(KERN_INFO "fpc1020a: User ID to delete =
%d\n", fpc1020a->u_id);
                  if(fpc1020a->u_id > 0 && fpc1020a->u_id < 99) {
                        fpc1020a_delete_user(fpc1020a->u_id);
                  } else {
                        printk("User ID invalid\n");
                  }
                  break;

            case DELETE_ALL_USER:
                  fpc1020a->tx_count = 0;
                  printk("fpc1020a: Deleting all users\n");
                  fpc1020a_delete_all();
                  break;


            case GET_FP_IMAGE:
                  fpc1020a->tx_count = 0;
                  // Reallocate memory for storing image data
                  TX_BUF_SIZE = 3211;
                  fpc1020a->tx_buf = krealloc(fpc1020a->tx_buf, 3211
* sizeof(uint8_t), GFP_KERNEL);


                  printk("fpc1020a: Getting user's fingeprint
image\n");
                  fpc1020a_get_image();
                  if(fpc1020a->rtflag == ACK_SUCCESS) {
                        printk(KERN_INFO "fpc1020a: Got the
fingerprint image\n");
                        if(copy_to_user((uint8_t*) arg, &(fpc1020a-
>img_matrix), sizeof(fpc1020a->img_matrix))) {
                              printk(KERN_ERR "fpc1020a: Couldn't
send image data to user space\n");
                        } else {
                              printk(KERN_INFO "fpc1020a: Sent image
data to user space\n");
                        }
                  } else {
                        printk(KERN_ERR "fpc1020a: Couldn't retrieve
image data\n");
                  }


                  TX_BUF_SIZE = 8;
                  fpc1020a->tx_buf = krealloc(fpc1020a->tx_buf, 8 *
sizeof(uint8_t), GFP_KERNEL);
                  break;


            default:
                  printk(KERN_INFO "Default\n");
                  break;
      }
      return 0;
}
```

```
/* FPC1020A functionality implementation */
static char fpc1020a_gen_check_sum(unsigned char wLen, unsigned char
*ptr)
{
	unsigned char i, temp = 0;

	for(i = 0; i < wLen; i++) {
		temp ^= *(ptr + i);
	}
	return temp;
}

static void fpc1020a_send_package(unsigned char wLen, unsigned char
*ptr)
{
	unsigned int i = 0, len = 0;

	*(fpc1020a->rx_buf) = DATA_START;

	for(i = 0; i < wLen; i++) {
		*(fpc1020a->rx_buf + 1 + i) =  *(ptr + i);
	}

	*(fpc1020a->rx_buf + wLen + 1) = fpc1020a_gen_check_sum(wLen,
ptr);
	*(fpc1020a->rx_buf + wLen + 2) = DATA_END;
	len = wLen + 3;

	// Add flag and UART send code here
	printk("pi: TX buffer:\n");
	for(i = 0; i < len; i++) {
		printk(KERN_CONT "0x%02x ", *(fpc1020a->rx_buf + i));
	}
	serdev_device_write_buf(fpc1020a->serdev, fpc1020a->rx_buf,
len);
	printk(KERN_INFO "pi: TX buffer sent to FPC1020A\n");
}
```

```
static char fpc1020a_check_package(unsigned char cmd)
{
      fpc1020a->rtflag = ACK_FAIL;

      wait_event_interruptible(wait_queue_tx, fpc1020a->tx_end_flag
== 1);

      switch(cmd){
            case CMD_ENROLL1:
            case CMD_ENROLL2:
            case CMD_ENROLL3:
                  if(ACK_SUCCESS == *(fpc1020a->tx_buf + 4)){
                        fpc1020a->rtflag = ACK_SUCCESS;
                  } else if(ACK_USER_EXIST == *(fpc1020a->tx_buf +
4)){
                        fpc1020a->rtflag = ACK_USER_EXIST;
                        msleep(500);
                  } else if(ACK_USER_OCCUPIED == *(fpc1020a->tx_buf
+ 4)){
                        fpc1020a->rtflag = ACK_USER_OCCUPIED;
                        msleep(500);
                  } else if(ACK_TIMEOUT == *(fpc1020a->tx_buf + 4)){
                        fpc1020a->rtflag = ACK_TIMEOUT;
                        msleep(500);
                  }
                  break;

            case CMD_DELETE_USER:
                  if(ACK_SUCCESS == *(fpc1020a->tx_buf + 4)){
                        fpc1020a->rtflag = ACK_SUCCESS;
                  }
                  break;

            case CMD_DELETE_ALL:
                  if(ACK_SUCCESS == *(fpc1020a->tx_buf + 4)){
                        fpc1020a->rtflag = ACK_SUCCESS;
```

```
                    }
                    break;

            case CMD_IDENTIFY:
                    if(ACK_SUCCESS == *(fpc1020a->tx_buf + 4)){
                          fpc1020a->rtflag = ACK_SUCCESS;
                    }
                    break;

            case CMD_USER_COUNT:
                    if(ACK_SUCCESS == *(fpc1020a->tx_buf + 4)){
                          fpc1020a->rtflag = ACK_SUCCESS;
                          fpc1020a->u_count = *(fpc1020a->tx_buf + 3);
                    }
                    break;

            case CMD_SEARCH:
                    if((1 == *(fpc1020a->tx_buf + 4)) || (2 ==
*(fpc1020a->tx_buf + 4)) || (3 == *(fpc1020a->tx_buf + 4))){
                          fpc1020a->rtflag = ACK_SUCCESS;
                          fpc1020a->u_id = *(fpc1020a->tx_buf + 3);
                    }
                    break;

            case CMD_GET_USER_ID:
                    if(ACK_SUCCESS == *(fpc1020a->tx_buf + 4)) {
                          fpc1020a->rtflag = ACK_SUCCESS;
                          fpc1020a->u_id = *(fpc1020a->tx_buf + 3);
                    }
                    break;

            case CMD_GET_IMAGE:
                    if(ACK_SUCCESS == *(fpc1020a->tx_buf + 4)) {
                          fpc1020a->rtflag = ACK_SUCCESS;
                          pos = 9;
                          for(i = 0; i < 80; i++) {
                                for(j = 0; j < 40; j++) {
                                      fpc1020a->img_matrix[i][j] =
  *(fpc1020a->tx_buf + pos);
                                      pos++;
                                }
                          }
                    }
                    break;

              default:
                    break;
        }
        fpc1020a->tx_end_flag = 0; // TX end flag reset
        return fpc1020a->rtflag;
  }

  static char fpc1020a_search(void)
  {
        unsigned char buf[5];

        *buf = CMD_SEARCH;
        *(buf + 1) = 0x00;
        *(buf + 2) = 0x00;
        *(buf + 3) = 0x00;
        *(buf + 4) = 0x00;

        fpc1020a_send_package(5, buf);
        return fpc1020a_check_package(CMD_SEARCH);
  }
  /*
  static void fpc1020a_identify(unsigned int u_id)
  {
        unsigned char buf[5];

        *buf = CMD_IDENTIFY;
        *(buf + 1) = u_id >> 8;
        *(buf + 2) = u_id & 0xff;
        *(buf + 3) = 0x00;
```

```
	*(buf + 4) = 0x00;

	fpc1020a_send_package(5, buf);
}
*/
static void fpc1020a_enroll1(unsigned int u_id)
{
	unsigned char buf[5];

	*buf = CMD_ENROLL1;
	*(buf + 1) = u_id >> 8;
	*(buf + 2) = u_id & 0xff;
	*(buf + 3) = 1;                    // User permission
	*(buf + 4) = 0x00;

	fpc1020a_send_package(5, buf);
}

static void fpc1020a_enroll2(unsigned int u_id)
{
	unsigned char buf[5];

	*buf = CMD_ENROLL2;
	*(buf + 1) = u_id >> 8;
	*(buf + 2) = u_id & 0xff;
	*(buf + 3) = 1;                    // User permission
	*(buf + 4) = 0x00;

	fpc1020a_send_package(5, buf);
}

static void fpc1020a_enroll3(unsigned int u_id)
{
	unsigned char buf[5];

	*buf = CMD_ENROLL3;
```

```
      *(buf + 1) = u_id >> 8;
      *(buf + 2) = u_id & 0xff;
      *(buf + 3) = 1;                     // User permission
      *(buf + 4) = 0x00;

      fpc1020a_send_package(5, buf);
}

static char fpc1020a_enroll(unsigned int u_id)
{
      printk("fpc1020a: Adding new fingerprint started\n");
      fpc1020a->tx_count = 0;
      fpc1020a_enroll1(u_id);
      fpc1020a_check_package(CMD_ENROLL1);
      printk("1st Enroll\n");
      if(fpc1020a->rtflag != ACK_SUCCESS) {
            return fpc1020a->rtflag;
      }
      printk("Remove finger for 2nd enroll\n");
      msleep(1000);

      fpc1020a->tx_count = 0;
      fpc1020a_enroll2(u_id);
      fpc1020a_check_package(CMD_ENROLL2);
      printk("2nd Enroll\n");
      if(fpc1020a->rtflag != ACK_SUCCESS) {
            return fpc1020a->rtflag;
      }
      printk("Remove finger for 3rd enroll\n");
      msleep(1000);

      fpc1020a->tx_count = 0;
      fpc1020a_enroll3(u_id);
      fpc1020a_check_package(CMD_ENROLL3);
      printk("3rd Enroll\n");
      if(fpc1020a->rtflag != ACK_SUCCESS) {
            return fpc1020a->rtflag;
      }
      printk("Scan Finished. Remove finger\n");
      msleep(1000);
      return fpc1020a->rtflag;
}

static char fpc1020a_delete_all(void)
{
      unsigned char buf[5];

      *buf = CMD_DELETE_ALL;
      *(buf + 1) = 0x00;
      *(buf + 2) = 0x00;
      *(buf + 3) = 0x00;
      *(buf + 4) = 0x00;

      fpc1020a_send_package(5, buf);
      return fpc1020a_check_package(CMD_DELETE_ALL);
}

static char fpc1020a_delete_user(unsigned int u_id)
{
      unsigned char buf[5];

      *buf = CMD_DELETE_USER;
      *(buf + 1) = u_id >> 8;
      *(buf + 2) = u_id & 0xFF;
      *(buf + 3) = 0x00;
      *(buf + 4) = 0x00;

      fpc1020a_send_package(5, buf);
      return fpc1020a_check_package(CMD_DELETE_USER);
}

static char fpc1020a_user_count(void)
```

```
{
      unsigned char buf[5];

      *buf = CMD_USER_COUNT;
      *(buf + 1) = 0x00;
      *(buf + 2) = 0x00;
      *(buf + 3) = 0x00;
      *(buf + 4) = 0x00;

      fpc1020a_send_package(5, buf);
      return fpc1020a_check_package(CMD_USER_COUNT);
}
/*
static char fpc1020a_get_user_id(void)
{
      unsigned char buf[5];

      *buf = CMD_GET_USER_ID;
      *(buf + 1) = 0x00;
      *(buf + 2) = 0x00;
      *(buf + 3) = 0x00;
      *(buf + 4) = 0x00;

      fpc1020a_send_package(5, buf);
      return fpc1020a_check_package(CMD_GET_USER_ID);
}
*/
static char fpc1020a_get_image(void)
{
      unsigned char buf[5];

      *buf = CMD_GET_IMAGE;
      *(buf + 1) = 0x00;
      *(buf + 2) = 0x00;
      *(buf + 3) = 0x00;
      *(buf + 4) = 0x00;
```

```
	fpc1020a_send_package(5, buf);
	return fpc1020a_check_package(CMD_GET_IMAGE);
}

static int __init fpc1020a_driver_init(void)
{
	temp_flag = alloc_chrdev_region(&dev, 0, 1, "fpc1020_driver");

	if(temp_flag < 0){
		printk(KERN_ERR "Cannot allocate major number for
device\n");
		return -1;
	}
	printk(KERN_NOTICE "Printing device major and minor\n");
	printk(KERN_INFO "Major = %d Minor = %d \n", MAJOR(dev),
MINOR(dev));

	cdev_init(&fpc1020a_cdev, &fops);
	temp_flag = cdev_add(&fpc1020a_cdev, dev, 1);
	if(temp_flag < 0){
		printk(KERN_ERR "Cannot add the device to the
system\n");
		goto r_class;
	}

	dev_class = class_create(THIS_MODULE, "fpc1020a_class");
	/*
	if(IS_ERR(class_create(THIS_MODULE, "fpc1020a_class"))){
		printk(KERN_ERR "Cannot create the struct class for the
device\n");
		goto r_class;
	*/

	if(IS_ERR(device_create(dev_class, NULL, dev, NULL,
"fpc1020a_dev"))){
		printk(KERN_ERR "Cannot create the device\n");
		goto r_device;
	}

	temp_flag = serdev_device_driver_register(&fpc1020a_driver);
	if(temp_flag){
		printk(KERN_ERR "fpc1020a - Error! Could not load
driver\n");
		return -1;
	}

	printk("fpc1020a - Loaded the driver...\n");
	return 0;

r_device:
	class_destroy(dev_class);
r_class:
	unregister_chrdev_region(dev, 1);
	return -1;
}

static void __exit fpc1020a_driver_exit(void)
{
	fpc1020a->tx_end_flag = 1;

	serdev_device_driver_unregister(&fpc1020a_driver);
	device_destroy(dev_class, dev);
	class_destroy(dev_class);
	cdev_del(&fpc1020a_cdev);
	unregister_chrdev_region(dev, 1);
	printk("fpc1020a - Unload driver\n");
}

module_init(fpc1020a_driver_init);
module_exit(fpc1020a_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("SIPLAB");
MODULE_DESCRIPTION("A simple loopback driver for an UART port");
```

```
MODULE_VERSION("1.0");
```

A3. Details of FPC1020A driver's Device Tree Overlay

```
/dts-v1/;
/plugin/;
/ {
	compatible = "brcm, bcm2835";
	fragment@0 {
		target = <&uart1>;
		__overlay__ {
			uartdev {
				compatible = "fpc,fpc1020a";
				current-speed = <19200>;
				pinctrl-names = "default";
				pinctrl-0 = <&uart1_pins>;
				status = "okay";
			};
		};
	};
};
```

A4. Details of FPC1020A driver’s Makefile

```
obj-m += fpc1020a_driver.o


KDIR = /lib/modules/$(shell uname -r)/build


all: module dt
	echo Built Device Tree Overlay and Kernel Module


module:
	make -C $(KDIR) M=$(shell pwd) modules


dt: fpc1020a_overlay.dts
	dtc -@ -I dts -O dtb -o fpc1020a_overlay.dtbo
fpc1020a_overlay.dts
```

```
clean:
      make -C $(KDIR) M=$(shell pwd) clean
      rm -rf fpc1020a_overlay.dtbo
```

A5. Detail of AT24C256 EEPROM's header file

```
#ifndef __AT24C256_H__
#define __AT24C256_H__

#define PAGE_SIZE 64
#define IMAGE_SIZE 504

#define BYTE_WRITE _IOW('g', 1, uint8_t*)
#define PAGE_WRITE _IOW('g', 2, uint8_t*)
#define CURRENT_READ _IOR('g', 3, uint8_t *)
#define SEQUENTIAL_READ _IOR('g', 4, uint8_t*)
//#define WRITE_IMAGE _IOW('g', 5, uint8_t*)
//#define READ_IMAGE _IOR('g', 6, uint8_t*)
#endif // __TESTNOKIA5110_H__
```

A6. Detail of AT24C256 EEPROM driver's source code

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/i2c.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <linux/kdev_t.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/uaccess.h>
#include <linux/miscdevice.h>
#include <linux/ioctl.h>
```

```
#include <linux/delay.h>
#include "at24c256.h"


#define DRV_LICENSE "GPL"
#define DRV_AUTHOR "Ton"
#define DRV_DESC "AT24C256 EEPROM"


static int at24c256_i2c_open(struct inode *inodes, struct file
*filp);
static int at24c256_i2c_release(struct inode *inodes, struct file
*filp);
static long at24c256_i2c_ioctl(struct file *file, unsigned int cmd,
unsigned long arg);
static ssize_t at24c256_i2c_write(struct file *filp, const char
__user *buffer, size_t size, loff_t *f_pos);
static ssize_t at24c256_i2c_read(struct file *filp, char __user
*buffer, size_t size, loff_t *f_pos);
static int at24c256_i2c_probe(struct i2c_client *client, const struct
i2c_device_id *id);
static int at24c256_i2c_remove(struct i2c_client *client);


int at24c256_byte_write(struct i2c_client *client, uint8_t
*user_data);
int at24c256_page_write(struct i2c_client *client, uint8_t
*user_data);
int at24c256_current_read(struct i2c_client *client, uint8_t
*user_data);
int at24c256_sequential_read(struct i2c_client *client, uint8_t
*user_data);
int at24c256_write_image(struct i2c_client *client, uint8_t *image);
int at24c256_read_image(struct i2c_client *client, uint8_t *image);


static const struct file_operations fops = {
    .owner = THIS_MODULE,
    .write = at24c256_i2c_write,
    .read = at24c256_i2c_read,
    .open = at24c256_i2c_open,
    .release = at24c256_i2c_release,
    .unlocked_ioctl = at24c256_i2c_ioctl,
};
int position;
int prev_position;
static struct at24c256
{
    struct miscdevice m_dev;
    struct i2c_client *client;
    uint8_t read_buffer[PAGE_SIZE];
    uint8_t write_buffer[PAGE_SIZE];
    uint8_t write_data;
    uint8_t read_data;
};

static int at24c256_i2c_open(struct inode *inodes, struct file *filp)
{
    struct at24c256 *at24c256 = container_of(filp->private_data,
struct at24c256, m_dev);
    if (at24c256)
        filp->private_data = at24c256;
    else
        return -ENODEV;
    pr_info("Open device file successfully!");
    return 0;
    pr_info("Open device file successfully!");
    return 0;
}

static int at24c256_i2c_release(struct inode *inodes, struct file
*filp)
{
    filp->private_data = NULL;
    pr_info("Close device file successfully!");
    return 0;
}

static long at24c256_i2c_ioctl(struct file *file, unsigned int cmd,
unsigned long arg)
```

```
{
    struct at24c256 *at24c256 = file->private_data;
    if (!at24c256)
        return -EFAULT;
    int ret;
    switch (cmd)
    {
    case BYTE_WRITE:
        if (copy_from_user(&(at24c256->write_data), (uint8_t *)arg, 
sizeof(at24c256->write_data)))
        {
            return -EFAULT;
        }

        ret = at24c256_byte_write(at24c256->client, at24c256-
>write_data);
        break;
    case PAGE_WRITE:
        if (copy_from_user(&(at24c256->write_buffer), (uint8_t *)arg, 
PAGE_SIZE))
        {
                return -EFAULT;
        }
        ret = at24c256_page_write(at24c256->client, at24c256-
>write_buffer);
        break;
    case CURRENT_READ:
        ret = at24c256_current_read(at24c256->client, &at24c256-
>read_data);
        if (ret > 0)
        {
            if (copy_to_user((uint8_t *)arg, &at24c256->read_data, 
sizeof(at24c256->read_data)))
            {
                return -EFAULT;
            }
        }
        break;
    case SEQUENTIAL_READ:
        ret = at24c256_sequential_read(at24c256->client, at24c256-
>read_buffer);
        if (ret > 0)
        {
            if (copy_to_user((uint8_t *)arg, &at24c256->read_buffer,
PAGE_SIZE))
            {
                return -EFAULT;
            }
        }
        break;
    default:
        return -ENOTTY; // Command not supported
    }
    if (ret < 0)
    {
        return ret;
    }
    printk("nokia5110 ioctl function\n");
    return 0;
}

static ssize_t at24c256_i2c_write(struct file *filp, const char
__user *buffer, size_t size, loff_t *f_pos)
{

    struct at24c256 *at24c256 = filp->private_data;
    if (!at24c256)
        return -EFAULT;
    int ret, tmp, i;
    struct i2c_msg msg;
    char temp_data[PAGE_SIZE];
    char buffer_data[PAGE_SIZE + 2];

    if (copy_from_user(temp_data, buffer, PAGE_SIZE) != 0)
    {
        return -EFAULT;
    };

    memset(buffer_data, 0, sizeof(buffer_data));
    buffer_data[1] = (char)0x02;
    buffer_data[0] = (char)0x00;

    for (i = 0; i < PAGE_SIZE; ++i)
    {
        buffer_data[i + 2] = temp_data[i];
    }

    msg.addr = at24c256->client->addr;
    msg.flags = at24c256->client->flags & 0;
    msg.buf = &buffer_data[0];
    msg.len = PAGE_SIZE + 2;

    ret = i2c_transfer(at24c256->client->adapter, &msg, 1);
    tmp = (ret == 1) ? msg.len : ret;

    printk("i2c code: %d return code: %d addr: 0x%02x%02x ", ret,
tmp, buffer_data[0], buffer_data[1]);

    for (i = 0; i < PAGE_SIZE; ++i)
    {
        printk("Write Data: 0x%02x", buffer_data[i + 2]);
    }

    return size;
}

static ssize_t at24c256_i2c_read(struct file *filp, char __user
*buffer, size_t size, loff_t *f_pos)
{
    struct at24c256 *at24c256 = filp->private_data;
    if (!at24c256)
        return -EFAULT;
    int ret, tmp, i;
    char word_addr[2];
    char buffer_data[PAGE_SIZE];
    struct i2c_msg msg[2];

    memset(word_addr, 0, sizeof(word_addr));
    memset(buffer_data, 0, sizeof(buffer_data));

    word_addr[1] = (char)0x02;
    word_addr[0] = (char)0x00;

    msg[0].addr = at24c256->client->addr;
    msg[0].flags = at24c256->client->flags & 0;
    msg[0].buf = &word_addr[0];
    msg[0].len = 2;

    msg[1].addr = at24c256->client->addr;
    msg[1].flags = I2C_M_RD;
    msg[1].buf = &buffer_data[0];
    msg[1].len = PAGE_SIZE;

    i2c_transfer(at24c256->client->adapter, msg, 2);
    tmp = (ret == 1) ? msg[1].len : ret;

    printk("i2c code: %d return code: %d addr: 0x%02x%02x ", ret,
tmp, word_addr[0], word_addr[1]);

    for (i = 0; i < PAGE_SIZE; ++i)
    {
        printk("Read data: 0x%02x", buffer_data[i]);
    }
    copy_to_user(buffer, buffer_data, PAGE_SIZE);
    return size;
}

int at24c256_current_read(struct i2c_client *client, uint8_t
*user_data)
```

```
{
    int ret, tmp, i;
    struct i2c_msg msg;

    msg.addr = client->addr;
    msg.flags = I2C_M_RD;
    msg.buf = user_data;
    msg.len = 1;
    ret = i2c_transfer(client->adapter, &msg, 1);
    if (ret < 0)
    {
        dev_err(&client->dev, "Failed to read from EEPROM: %d\n",
ret);
        return ret;
    }
    tmp = (ret == 1) ? msg.len : ret;
        printk("Read data: 0x%02x", *user_data);
    prev_position++;
    return ret;
}

int at24c256_sequential_read(struct i2c_client *client, uint8_t
*user_data)
{
    int ret, tmp, i;
    uint8_t word_addr[2];
    struct i2c_msg msg[2];

    word_addr[0] = (u8)(prev_position >> 8) & 0xFF;
    word_addr[1] = (u8)(prev_position & 0xFF);

    msg[0].addr = client->addr;
    msg[0].flags = client->flags & 0;
    msg[0].buf = word_addr;
    msg[0].len = 2;

    msg[1].addr = client->addr;
    msg[1].flags = I2C_M_RD;
    msg[1].buf = user_data;
    msg[1].len = PAGE_SIZE;

    ret = i2c_transfer(client->adapter, msg, 2);
    if (ret < 0)
    {
        dev_err(&client->dev, "Failed to read from EEPROM: %d\n",
ret);
        return ret;
    }
    tmp = (ret == 1) ? msg[1].len : ret;

    printk("i2c code: %d return code: %d addr: 0x%02x%02x ", ret,
tmp, word_addr[0], word_addr[1]);

    for (i = 0; i < PAGE_SIZE; ++i)
    {
        printk("Read data: 0x%02x", *(user_data+i));
    }
    prev_position += PAGE_SIZE;
    return ret;
}

int at24c256_byte_write(struct i2c_client *client, uint8_t
*user_data)
{
    int ret, tmp, i;
    struct i2c_msg msg;
    uint8_t buffer_data[3];
    memset(buffer_data, 0, sizeof(buffer_data));

    buffer_data[0] = (u8)(position >> 8) & 0xFF;
    buffer_data[1] = (u8)(position & 0xFF);
    buffer_data[2] = user_data;

    msg.addr = client->addr;
    msg.flags = client->flags & 0;
    msg.buf = buffer_data;
    msg.len = 1;

    ret = i2c_transfer(client->adapter, &msg, 1);
    if (ret < 0)
    {
        dev_err(&client->dev, "Failed to write from EEPROM: %d\n",
ret);
        return ret;
    }
    tmp = (ret == 1) ? msg.len : ret;

    printk("i2c code: %d return code: %d addr: 0x%02x%02x ", ret,
tmp, buffer_data[0], buffer_data[1]);
    printk("Write Data: 0x%02x", buffer_data[2]);
    prev_position = position;
    position++;
    return ret;
}

int at24c256_page_write(struct i2c_client *client, uint8_t
*user_data)
{
    int ret, tmp, i;
    struct i2c_msg msg;
    uint8_t buffer_data[PAGE_SIZE + 2];
    memset(buffer_data, 0, sizeof(buffer_data));

    buffer_data[0] = (u8)(position >> 8) & 0xFF;
    buffer_data[1] = (u8)(position & 0xFF);

    for (i = 0; i < PAGE_SIZE; ++i)
    {
        buffer_data[i + 2] = *(user_data+i);
    }
    msg.addr = client->addr;

    msg.flags = client->flags & 0;

    msg.buf = buffer_data;

    msg.len = PAGE_SIZE + 2;


    ret = i2c_transfer(client->adapter, &msg, 1);

    if (ret < 0)

    {

        dev_err(&client->dev, "Failed to write from EEPROM: %d\n",
ret);

        return ret;

    }

    tmp = (ret == 1) ? msg.len : ret;


    printk("i2c code: %d return code: %d addr: 0x%02x%02x ", ret,
tmp, buffer_data[0], buffer_data[1]);


    for (i = 0; i < PAGE_SIZE; ++i)

    {

        printk("Write Data: 0x%02x", buffer_data[i + 2]);

    }

    prev_position = position;

    position += PAGE_SIZE;

    return ret;

}


static int at24c256_i2c_probe(struct i2c_client *client, const struct
i2c_device_id *id)

{

    int ret;

    struct at24c256 *at24c256 = NULL;

    at24c256 = kzalloc(sizeof(*at24c256), GFP_KERNEL);

    if (!at24c256)

    {

        ret = -ENOMEM;

        kfree(at24c256);

        return ret;
    };
    at24c256->client = client;
    i2c_set_clientdata(client, at24c256);

    at24c256->m_dev.minor = MISC_DYNAMIC_MINOR;
    at24c256->m_dev.name = "at24c256";
    at24c256->m_dev.mode = 0666;
    at24c256->m_dev.fops = &fops;
    printk(KERN_INFO "Probed successfully!");
    return misc_register(&at24c256->m_dev);
}

static int at24c256_i2c_remove(struct i2c_client *client)
{
    printk("\tRemove at24c256 device...\n");
    struct at24c256 *at24c256 = i2c_get_clientdata(client);
    if (!at24c256)
    {
        return -1;
    }
    else
    {
        misc_deregister(&at24c256->m_dev);
        kfree(at24c256);
        printk("\tRemove at24c256 device success\n");
    }
    return 0;
}
static const struct of_device_id at24c256_i2c_of_match[] = {
    {.compatible = "at24c256"},
    {}};

MODULE_DEVICE_TABLE(of, at24c256_i2c_of_match);

static struct i2c_driver at24c256_driver = {
    .driver = {
        .name = "at24c256",
        .owner = THIS_MODULE,
        .of_match_table = at24c256_i2c_of_match,
    },
    .probe = at24c256_i2c_probe,
    .remove = at24c256_i2c_remove,
};

module_i2c_driver(at24c256_driver);

MODULE_LICENSE(DRV_LICENSE);
MODULE_AUTHOR(DRV_AUTHOR);
MODULE_DESCRIPTION(DRV_DESC);
```

A7. Detail of AT24C256 driver's Device Node

```
&i2c2 {
      status = "okay";

      at24c256: at24c256@50 {
            compatible = "at24c256";
            reg = <0x50>;
            status = "okay";
      };
};
```

A8. Detail of AT24C256 driver's Makefile

```
BBB_KERNEL :=
/home/ton/Linux_Programming/BBB/kernelbuildscripts/KERNEL
TOOLCHAIN :=
/home/ton/Linux_Programming/BBB/kernelbuildscripts/dl/gcc-8.5.0-
nolibc/arm-linux-gnueabi/bin/arm-linux-gnueabi-

EXTRA_CFLAGS=-Wall
obj-m := at24c256.o

all:
      make ARCH=arm CROSS_COMPILE=$(TOOLCHAIN) -C $(BBB_KERNEL)
M=$(shell pwd) modules

clean:

      make -C $(BBB_KERNEL) M=$(shell pwd) clean
```

A9. Detail of LCD NOKIA 5110 driver's header file

```
#ifndef __NOKIA5110_H__

#define __NOKIA5110_H_


#define LCD_WIDTH 84

#define LCD_HEIGHT 48


#define LCD_CMD 0

#define LCD_DATA 1


#define Y_BASE_COR 0x40

#define X_BASE_COR 0x80


#define BASIC_INSTRUCT 0x20           // H = 0 V = 0

#define EXTEND_INSTRUCT 0x21          // H = 1 V = 0

#define BASIC_INSTRUCT_VERTICAL 0x22  // H = 0 V = 1

#define EXTEND_INSTRUCT_VERTICAL 0x23 // H = 1 V = 1


// if H = 0

#define DISPLAY_BLANK 0x08    // DE = 00

#define NORMAL_MODE 0x0C      // DE = 10

#define DISPLAY_SEG_ON 0x09   // DE = 01

#define INVERSE_VID_MODE 0x0D // DE = 11


// if H = 1

#define TEMP_CTRL 0x04 // Temp coef 0

#define BIAS_SYS 0x13  // Mux Rate 1 : 48


#define MAX_X 83

#define MAX_Y 5
```

```
struct position {
    uint8_t x;
    uint8_t y;
};

#define NEXT_LINE   _IO('a', 1)
#define GOTO_XY     _IOW('a', 2, struct position*)
#define CLEAR       _IO('a', 3)
#define PRINT_IMAGE _IOW('a', 4, uint8_t*)

static const uint8_t ASCII[92][6] = {
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, // space
    {0x00, 0x00, 0x00, 0x2f, 0x00, 0x00}, // !
    {0x00, 0x00, 0x07, 0x00, 0x07, 0x00}, // "
    {0x00, 0x14, 0x7f, 0x14, 0x7f, 0x14}, // #
    {0x00, 0x24, 0x2a, 0x7f, 0x2a, 0x12}, // $
    {0x00, 0x62, 0x64, 0x08, 0x13, 0x23}, // %
    {0x00, 0x36, 0x49, 0x55, 0x22, 0x50}, // &
    {0x00, 0x00, 0x05, 0x03, 0x00, 0x00}, // '
    {0x00, 0x00, 0x1c, 0x22, 0x41, 0x00}, // (
    {0x00, 0x00, 0x41, 0x22, 0x1c, 0x00}, // )
    {0x00, 0x14, 0x08, 0x3E, 0x08, 0x14}, // *
    {0x00, 0x08, 0x08, 0x3E, 0x08, 0x08}, // +
    {0x00, 0x00, 0x00, 0xA0, 0x60, 0x00}, // ,
    {0x00, 0x08, 0x08, 0x08, 0x08, 0x08}, // -
    {0x00, 0x00, 0x60, 0x60, 0x00, 0x00}, // .
    {0x00, 0x20, 0x10, 0x08, 0x04, 0x02}, // /
    {0x00, 0x3E, 0x51, 0x49, 0x45, 0x3E}, // 0
    {0x00, 0x00, 0x42, 0x7F, 0x40, 0x00}, // 1
    {0x00, 0x42, 0x61, 0x51, 0x49, 0x46}, // 2
    {0x00, 0x21, 0x41, 0x45, 0x4B, 0x31}, // 3
    {0x00, 0x18, 0x14, 0x12, 0x7F, 0x10}, // 4
    {0x00, 0x27, 0x45, 0x45, 0x45, 0x39}, // 5
    {0x00, 0x3C, 0x4A, 0x49, 0x49, 0x30}, // 6
    {0x00, 0x01, 0x71, 0x09, 0x05, 0x03}, // 7
    {0x00, 0x36, 0x49, 0x49, 0x49, 0x36}, // 8
    {0x00, 0x06, 0x49, 0x49, 0x29, 0x1E}, // 9
    {0x00, 0x00, 0x36, 0x36, 0x00, 0x00}, // :
    {0x00, 0x00, 0x56, 0x36, 0x00, 0x00}, // ;
    {0x00, 0x08, 0x14, 0x22, 0x41, 0x00}, // <
    {0x00, 0x14, 0x14, 0x14, 0x14, 0x14}, // =
    {0x00, 0x00, 0x41, 0x22, 0x14, 0x14}, // >
    {0x00, 0x02, 0x01, 0x51, 0x09, 0x06}, // ?
    {0x00, 0x32, 0x49, 0x59, 0x51, 0x3E}, // @
    {0x00, 0x7C, 0x12, 0x11, 0x12, 0x7C}, // A
    {0x00, 0x7F, 0x49, 0x59, 0x51, 0x3E}, // B
    {0x00, 0x3E, 0x41, 0x41, 0x41, 0x22}, // C
    {0x00, 0x7F, 0x41, 0x41, 0x22, 0x1C}, // D
    {0x00, 0x7F, 0x49, 0x49, 0x49, 0x41}, // E
    {0x00, 0x7F, 0x09, 0x09, 0x09, 0x01}, // F
    {0x00, 0x3E, 0x41, 0x49, 0x49, 0x7A}, // G
    {0x00, 0x7F, 0x08, 0x08, 0x08, 0x7F}, // H
    {0x00, 0x00, 0x41, 0x7F, 0x41, 0x00}, // I
    {0x00, 0x20, 0x40, 0x41, 0x3F, 0x01}, // J
    {0x00, 0x7F, 0x08, 0x14, 0x22, 0x41}, // K
    {0x00, 0x7F, 0x40, 0x40, 0x40, 0x40}, // L
    {0x00, 0x7F, 0x02, 0x0C, 0x02, 0x7F}, // M
    {0x00, 0x7F, 0x04, 0x08, 0x10, 0x7F}, // N
    {0x00, 0x3E, 0x41, 0x41, 0x41, 0x3E}, // O
    {0x00, 0x7F, 0x09, 0x09, 0x09, 0x06}, // P
    {0x00, 0x3E, 0x41, 0x51, 0x21, 0x5E}, // Q
    {0x00, 0x7F, 0x09, 0x19, 0x29, 0x46}, // R
    {0x00, 0x46, 0x49, 0x49, 0x49, 0x31}, // S
    {0x00, 0x01, 0x01, 0x7F, 0x01, 0x01}, // T
    {0x00, 0x3F, 0x40, 0x40, 0x40, 0x3F}, // U
    {0x00, 0x1F, 0x20, 0x40, 0x20, 0x1F}, // V
    {0x00, 0x3F, 0x40, 0x38, 0x40, 0x3F}, // W
    {0x00, 0x63, 0x14, 0x08, 0x14, 0x63}, // X
    {0x00, 0x07, 0x08, 0x70, 0x08, 0x07}, // Y
    {0x00, 0x61, 0x51, 0x49, 0x45, 0x43}, // Z
    {0x00, 0x00, 0x7F, 0x41, 0x41, 0x00}, // [
    {0x00, 0x55, 0x2A, 0x55, 0x2A, 0x55}, // 55
    {0x00, 0x00, 0x41, 0x41, 0x7F, 0x00}, // ]
    {0x00, 0x04, 0x02, 0x01, 0x02, 0x04}, // ^
    {0x00, 0x40, 0x40, 0x40, 0x40, 0x40}, // _
    {0x00, 0x00, 0x01, 0x02, 0x04, 0x00}, // '
    {0x00, 0x20, 0x54, 0x54, 0x54, 0x78}, // a
    {0x00, 0x7F, 0x48, 0x44, 0x44, 0x38}, // b
    {0x00, 0x38, 0x44, 0x44, 0x44, 0x20}, // c
    {0x00, 0x38, 0x44, 0x44, 0x48, 0x7F}, // d
    {0x00, 0x38, 0x54, 0x54, 0x54, 0x18}, // e
    {0x00, 0x08, 0x7E, 0x09, 0x01, 0x02}, // f
    {0x00, 0x18, 0xA4, 0xA4, 0xA4, 0x7C}, // g
    {0x00, 0x7F, 0x08, 0x04, 0x04, 0x78}, // h
    {0x00, 0x00, 0x44, 0x7D, 0x40, 0x00}, // i
    {0x00, 0x40, 0x80, 0x84, 0x7D, 0x00}, // j
    {0x00, 0x7F, 0x10, 0x28, 0x44, 0x00}, // k
    {0x00, 0x00, 0x41, 0x7F, 0x40, 0x00}, // l
    {0x00, 0x7C, 0x04, 0x18, 0x04, 0x78}, // m
    {0x00, 0x7C, 0x08, 0x04, 0x04, 0x78}, // n
    {0x00, 0x38, 0x44, 0x44, 0x44, 0x38}, // o
    {0x00, 0xFC, 0x24, 0x24, 0x24, 0x18}, // p
    {0x00, 0x18, 0x24, 0x24, 0x18, 0xFC}, // q
    {0x00, 0x7C, 0x08, 0x04, 0x04, 0x08}, // r
    {0x00, 0x48, 0x54, 0x54, 0x54, 0x20}, // s
    {0x00, 0x04, 0x3F, 0x44, 0x40, 0x20}, // t
    {0x00, 0x3C, 0x40, 0x40, 0x20, 0x7C}, // u
    {0x00, 0x1C, 0x20, 0x40, 0x20, 0x1C}, // v
    {0x00, 0x3C, 0x40, 0x30, 0x40, 0x3C}, // w
    {0x00, 0x44, 0x28, 0x10, 0x28, 0x44}, // x
    {0x00, 0x1C, 0xA0, 0xA0, 0xA0, 0x7C}, // y
    {0x00, 0x44, 0x64, 0x54, 0x4C, 0x44}, // z
    {0x14, 0x14, 0x14, 0x14, 0x14, 0x14}, // horizontal lines
};
#endif
```

A10. Detail of LCD NOKIA 5110 driver's source code

```
#include <linux/device.h>
```

```
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/types.h>
#include <linux/io.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/ioctl.h>
#include <linux/mm.h>
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/delay.h>
#include <linux/spi/spi.h>
#include <linux/string.h>
#include <linux/miscdevice.h>
#include "nokia5110.h"
#include "testnokia5110.h"

#define DRV_LICENSE "GPL"
#define DRV_AUTHOR "Ton"
#define DRV_DESC "LCD NOKIA5110"
#define IMG_SIZE 504

static struct nokia5110
{
    uint8_t *recv_buffer;
    struct miscdevice m_dev;
    struct spi_device *nokia5110_spi;
    uint8_t current_X;
    uint8_t current_Y;
};

static int re_pin = 30;
static int dc_pin = 31;
static int be_pin = 48;
static int nokia5110_spi_open(struct inode *inodes, struct file
*filp);

static int nokia5110_spi_release(struct inode *inodes, struct file
*filp);

static long nokia5110_spi_ioctl(struct file *file, unsigned int cmd,
unsigned long arg);

static ssize_t nokia5110_spi_write(struct file *filp, const char
__user *buffer, size_t size, loff_t *f_pos);

static int nokia5110_probe(struct spi_device *spi);

static int nokia5110_remove(struct spi_device *spi);


void nokia5110_init(struct spi_device *spi);

void nokia5110_clear_screen(struct nokia5110 *nokia5110);

void nokia5110_send_byte(struct spi_device *spi, bool, uint8_t);

void nokia5110_write_char(struct spi_device *spi, uint8_t);

void nokia5110_write_string(struct spi_device *spi, uint8_t *);

void nokia5110_set_XY_coor(struct nokia5110 *nokia5110, uint8_t,
uint8_t);

void nokia5110_next_line(struct nokia5110 *nokia5110);

void nokia5110_print_image(struct nokia5110 *nokia5110, uint8_t *);


static const struct file_operations fops = {

    .owner = THIS_MODULE,

    .write = nokia5110_spi_write,

    .open = nokia5110_spi_open,

    .release = nokia5110_spi_release,

    .unlocked_ioctl = nokia5110_spi_ioctl,

};


static int nokia5110_spi_open(struct inode *inodes, struct file
*filp)

{

    struct nokia5110 *nokia5110 = container_of(filp->private_data,
struct nokia5110, m_dev);

    if (nokia5110)

        filp->private_data = nokia5110;

    else

        return -ENODEV;
    pr_info("Open device file successfully!");
    return 0;
}

static int nokia5110_spi_release(struct inode *inodes, struct file
*filp)
{
    filp->private_data = NULL;
    pr_info("Close device file successfully!");
    return 0;
}

static long nokia5110_spi_ioctl(struct file *file, unsigned int cmd,
unsigned long arg)
{
    struct nokia5110 *nokia5110 = file->private_data;
    struct position pos;
    if (!nokia5110)
        return -EFAULT;
    switch (cmd)
    {
    case NEXT_LINE:
        nokia5110_next_line(nokia5110);
        break;
    case GOTO_XY:
        if (!copy_from_user(&pos, (struct position *)arg,
sizeof(struct position)))
            nokia5110_set_XY_coor(nokia5110, pos.x, pos.y);
        break;
    case CLEAR:
        nokia5110_clear_screen(nokia5110);
        break;
    case PRINT_IMAGE:
        nokia5110->recv_buffer = kzalloc(IMG_SIZE, GFP_KERNEL);
        if (copy_from_user(nokia5110->recv_buffer, (uint8_t *)arg,
IMG_SIZE) != 0)
        {
            kfree(nokia5110->recv_buffer);
            return -EFAULT;
        }
        nokia5110_print_image(nokia5110, nokia5110->recv_buffer);
        kfree(nokia5110->recv_buffer);
        return IMG_SIZE;
    default:
        printk("No valid cmd!");
        break;
    };
    printk("nokia5110 ioctl function\n");
    return 0;
}

static ssize_t nokia5110_spi_write(struct file *filp, const char
__user *buffer, size_t size, loff_t *f_pos)
{
    struct nokia5110 *nokia5110 = filp->private_data;
    if (!nokia5110)
        return -EFAULT;
    nokia5110->recv_buffer = kzalloc(size, GFP_KERNEL);
    if (copy_from_user(nokia5110->recv_buffer, buffer, size) != 0)
    {
        kfree(nokia5110->recv_buffer);
        return -EFAULT;
    }

    nokia5110_write_string(nokia5110->nokia5110_spi, nokia5110-
>recv_buffer);
    kfree(nokia5110->recv_buffer);
    return size;
}

void nokia5110_init(struct spi_device *spi)
{
    struct nokia5110 *nokia5110 = spi_get_drvdata(spi);
    // Set GPIOS
    gpio_set_value(re_pin, 0);
    udelay(2);
    gpio_set_value(re_pin, 1);
    udelay(2);
    gpio_set_value(be_pin, 1);

    // init LCD
    nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_CMD,
EXTEND_INSTRUCT); // LCD Extended Commands
    nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_CMD, 0xb1);
// Set LCD Cop (Contrast) //0xb1
    nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_CMD,
TEMP_CTRL);       // Set Temp Coefficent    //0x04
    nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_CMD, BIAS_SYS);
// LCD bias mode 1:48     //0x13
    nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_CMD,
BASIC_INSTRUCT);
    nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_CMD,
NORMAL_MODE);

    nokia5110_set_XY_coor(nokia5110, 0, 2);
    nokia5110_write_string(nokia5110->nokia5110_spi, "Hello
World!!!!");
    printk(KERN_INFO "Init successfully!");

    gpio_set_value(be_pin, 1);
}

void nokia5110_send_byte(struct spi_device *spi, bool is_cmd, uint8_t
data)
{
    if (is_cmd)
        gpio_set_value(dc_pin, 1);
    else
        gpio_set_value(dc_pin, 0);

    spi_write(spi, &data, sizeof(data)); // PASSING SPI DEVICE AS A
PARAM
}
```

```
void nokia5110_clear_screen(struct nokia5110 *nokia5110)
{
    int i;
    for (i = 0; i < LCD_WIDTH * LCD_HEIGHT / 8; ++i)
    {
        nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_DATA, 
0x00);
    }
    nokia5110_set_XY_coor(nokia5110, 0, 0); //
}


void nokia5110_write_char(struct spi_device *spi, uint8_t data)
{
    nokia5110_send_byte(spi, LCD_DATA, 0x00);
    int i;
    for (i = 0; i < 6; ++i)
        nokia5110_send_byte(spi, LCD_DATA, ASCII[data - 0x20][i]);
}


void nokia5110_write_string(struct spi_device *spi, uint8_t *data)
{
    while (*data)
    {
        nokia5110_write_char(spi, *data++);
    }
}
void nokia5110_print_image(struct nokia5110 *nokia5110, uint8_t 
*data)
{
    int i;
    for (i = 0; i < IMG_SIZE; ++i)
    {
        nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_DATA, 
data[i]);
    }
}
static int nokia5110_probe(struct spi_device *spi)
```

```
{
    int ret;
    struct nokia5110 *nokia5110 = NULL;
    nokia5110 = kzalloc(sizeof(*nokia5110), GFP_KERNEL);
    if (!nokia5110)
    {
        ret = -ENOMEM;
        kfree(nokia5110);
        return ret;
    }
    nokia5110->nokia5110_spi = spi;
    spi_set_drvdata(spi, nokia5110);
    nokia5110->m_dev.minor = MISC_DYNAMIC_MINOR;
    nokia5110->m_dev.name = "nokia5110";
    nokia5110->m_dev.mode = 0666;
    nokia5110->m_dev.fops = &fops;
    if (misc_register(&nokia5110->m_dev) < 0)
    {
        printk(KERN_ERR "Misc Register failed\n");
        return -1;
    }
    gpio_request(re_pin, "RE");
    gpio_request(dc_pin, "DC");
    gpio_request(be_pin, "BE");
    gpio_direction_output(re_pin, 0);
    gpio_direction_output(dc_pin, 0);
    gpio_direction_output(be_pin, 0);

    nokia5110_init(nokia5110->nokia5110_spi);
    nokia5110_clear_screen(nokia5110);
    nokia5110_set_XY_coor(nokia5110, 0, 2);
    nokia5110_write_string(nokia5110->nokia5110_spi, "Hello
World!!!!");
    printk(KERN_INFO "Probed successfully!");
    return 0;
}
```

```
static int nokia5110_remove(struct spi_device *spi)
{
    struct nokia5110 *nokia5110 = spi_get_drvdata(spi);
    if (!nokia5110)
    {
        return -1;
    }
    else
    {
        nokia5110_clear_screen(nokia5110);
        misc_deregister(&nokia5110->m_dev);
        kfree(nokia5110);
        printk("\tRemove nokia5110 device successfully!\n");
    };
    return 0;
}

void nokia5110_set_XY_coor(struct nokia5110 *nokia5110, uint8_t X,
uint8_t Y)
{
    nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_CMD, Y_BASE_COR
| Y);
    nokia5110_send_byte(nokia5110->nokia5110_spi, LCD_CMD, X_BASE_COR
| X * 6);
    nokia5110->current_X = X;
    nokia5110->current_Y = Y;
}

void nokia5110_next_line(struct nokia5110 *nokia5110)
{
    nokia5110->current_Y = (nokia5110->current_Y == MAX_Y) ? 0 :
(nokia5110->current_Y + 1);
    nokia5110_set_XY_coor(nokia5110, 0, nokia5110->current_Y);
}
struct of_device_id nokia5110_of_match[] = {
    {.compatible = "nokia5110"},
    {}};
```

```
MODULE_DEVICE_TABLE(of, nokia5110_of_match);

static struct spi_driver my_spi_driver = {
    .probe = nokia5110_probe,
    .remove = nokia5110_remove,
    .driver = {
        .name = "nokia5110",
        .owner = THIS_MODULE,
        .of_match_table = nokia5110_of_match,
    },
};

static int __init func_init(void)
{
    return spi_register_driver(&my_spi_driver);
}

static void __exit func_exit(void)
{
    return spi_unregister_driver(&my_spi_driver);
}

module_init(func_init);
module_exit(func_exit);

MODULE_LICENSE(DRV_LICENSE);
MODULE_AUTHOR(DRV_AUTHOR);
MODULE_DESCRIPTION(DRV_DESC);
```

A11. Detail of LCD NOKIA 5110 driver's Device Node

```
&am33xx_pinmux {
      bb_spi1_pins: pinmux_bb_spi1_pins {
            pinctrl-single,pins = <
                  AM33XX_PADCONF(AM335X_PIN_MCASP0_ACLKX, PIN_INPUT,
MUX_MODE3)  /* P9_31 (A13) mcasp0_aclkx.spi1_sclk */
                  AM33XX_PADCONF(AM335X_PIN_MCASP0_FSX, PIN_INPUT,
MUX_MODE3)  /* P9_29 (B13) mcasp0_fsx.spi1_d0 */
                  AM33XX_PADCONF(AM335X_PIN_MCASP0_AXR0, PIN_INPUT,
MUX_MODE3)  /* P9_30 (D12) mcasp0_axr0.spi1_d1 */
                  AM33XX_PADCONF(AM335X_PIN_MCASP0_AHCLKR, PIN_INPUT,
MUX_MODE3)  /* P9_28 (C12) mcasp0_ahclkr.spi1_cs0 */
            >;
      };
};

&spi1 {
      status = "okay";
      pinctrl-0 = <&bb_spi1_pins>;
      pinctrl-names = "default";
      cs-gpios = <&gpio3 19 1>;

      nokia5110: nokia5110@0 {
            compatible = "nokia5110";
            spi-max-frequency = <200000>;
            reg = <0>;
            status = "okay";
      };
};
```

A12. Detail of LCD NOKIA 5110 driver's Makefile

```
BBB_KERNEL :=
/home/ton/Linux_Programming/BBB/kernelbuildscripts/KERNEL
TOOLCHAIN :=
/home/ton/Linux_Programming/BBB/kernelbuildscripts/dl/gcc-8.5.0-
nolibc/arm-linux-gnueabi/bin/arm-linux-gnueabi-


EXTRA_CFLAGS=-Wall
obj-m := nokia5110.o


all:
      make ARCH=arm CROSS_COMPILE=$(TOOLCHAIN) -C $(BBB_KERNEL)
M=$(shell pwd) modules
```

```
clean:

      make -C $(BBB_KERNEL) M=$(shell pwd) clean
```