

John Madiou

Linux Device Drivers Development

Develop customized drivers for embedded Linux



Packt>

Linux Device Drivers Development

Develop customized drivers for embedded Linux

John Madieu



BIRMINGHAM - MUMBAI

Linux Device Drivers Development

Copyright © 2017 Packt Publishing All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2017

Production reference: 1111017

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78528-000-9

www.packtpub.com

Credits

Author John Madieu	Copy Editors Juliana Nair Safis Editing
Reviewer Jérôme Pouiller	Project Coordinator Judie Jose
Commissioning Editor Gebin George	Proofreader Safis Editing
Acquisition Editor Gebin George	Indexer Rekha Nair
Content Development Editor Devika Battike	Graphics Kirk D'Penha
Technical Editor Swathy Mohan	Production Coordinator Arvindkumar Gupta

About the Author

John Madieu is an embedded Linux and kernel engineer living in France, in Paris. His main activities consist of developing drivers and Board Support Packages (BSP) for companies in domains such as automation, transport, healthcare, energy, and the military. John works at EXPEMB, a French company that is a pioneer in electronical board design based on computer-on-module, and in embedded Linux solutions. He is an open source and embedded systems enthusiast, convinced that it is only by sharing knowledge that one learns more.

He is passionate about boxing, which he practised for 6 years professionally, and continues to transmit this passion through sessions of training that he provides voluntarily.

I would like to thank Devika Battike, Gebin George, and all the Packt team for their efforts to release this book on time. They are the people without whom this book would probably never have seen the light of day. It was a pleasure to work with them.

Finally, I would like to thank all the mentors I have had over the years, and who still continue to accompany me. Mentors such as Cyprien Pacôme Nguéjack for his programming skills that I have learned over the years, Jérôme Pouillier and Christophe Nowicki for introducing me buildroot and leading me to kernel programming, Jean-Christian Rerat and Jean-Philippe DU-Teil of EXPEMB for their coaching and accompaniment in my professional career; to all those I could not mention, I wish to thank them for having transmitted these connoises to me, which I have tried to disseminate through this book.

About the Reviewer

Jérôme Pouiller is a true geek and fascinated by understanding how things do work.

He was an early adopter of Linux. He found in Linux a system with no limits, where everything could be changed. Linux has provided an excellent platform to hack anything.

He graduated in machine learning at Ecole Pour l'Informatique et les Technologies Avancées (EPITA). Beside his studies, he learned electronics by himself. He quickly turned his attention to the piece of software at crossroad of all advanced systems: the operating system. It is now one of his favorite subjects.

For 15 years now, Jérôme Pouiller has designed (and often debugged) Linux firmware for a variety of industries (multimedia, healthcare, nuclear, military).

In addition to his consulting activities, Jérôme Pouiller is professor of operating systems at Institut National des Sciences Appliquées (INSA). He has written many course materials about system programming, operating system design, realtime systems, and more.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1785280007>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

I would like to thank my girlfriend for her support and all the sleepless nights accompanying the writing of this book, as well as Brigitte and François, my dear parents, for whom I have a thought and to whom I dedicate this book entirely.

- John Madieu

I would like to dedicate this book in the memory of my father, who left too.

- Jérôme Pouiller

Table of Contents

Preface

What this book covers

What you need for this book

Who this book is for

Conventions

Reader feedback

Customer support

Downloading the example code

Downloading the color images of this book

Errata

Piracy

Questions

1. Introduction to Kernel Development

Environment setup

Getting the sources

Source organization

Kernel configuration

Build your kernel

Kernel habits

Coding style

Kernel structures allocation/initialization

Classes, objects, and OOP

Summary

2. Device Driver Basis

- User space and kernel space
- The concept of modules

- Module dependencies
- depmod utility

- Module loading and unloading
- Manual loading
- modprobe and insmod

- /etc/modules-load.d/<filename>.conf

- Auto-loading

- Module unload

- Driver skeletons

- Module entry and exit point
- __init and __exit attributes

- Module information
- Licensing

- Module author(s)

- Module description

- Errors and message printing
- Error handling

- Handling null pointer errors

- Message printing – printk()

- Module parameters

- Building your first module
- The module's makefile

- In the kernel tree

- Out of the tree

- Building the module

- Summary

3. Kernel Facilities and Helper Functions

- Understanding container_of macro

- Linked lists

 - Creating and initializing the list

 - Dynamic method

 - Static method

 - Creating a list node

 - Adding a list node

 - Deleting a node from the list

 - Linked list traversal

- Kernel sleeping mechanism

 - Wait queue

- Delay and timer management

 - Standard timers

 - Jiffies and HZ

 - Timers API

 - Timer setup initialization

 - Standard timer example

 - High resolution timers (HRTs)

 - HRT API

 - HRT setup initialization

 - Dynamic tick/tickless kernel

 - Delays and sleep in the kernel

 - Atomic context

 - Nonatomic context

- Kernel locking mechanism

 - Mutex

 - Mutex API

 - Declare

 - Acquire and release

- Spinlock
 - Spinlock versus mutexes

- Work deferring mechanism
 - Softirqs and ksoftirqd
 - ksoftirqd

- Tasklets
 - Declaring a tasklet
 - Enabling and disabling a tasklet

- Tasklet scheduling

- Work queues
 - Kernel-global workqueue – the shared queue

- Dedicated work queue
 - Programming syntax

- Predefined (shared) workqueue and standard workqueue functions

- Kernel threads

- Kernel interruption mechanism
 - Registering an interrupt handler
 - Interrupt handler and lock

- Concept of bottom halves
 - The problem – interrupt handler design limitations

- The solution – bottom halves

- Tasklets as bottom halves

- Workqueue as bottom halves

- Softirqs as bottom half

- Threaded IRQs
 - Threaded bottom half

- Invoking user-space applications from the kernel

- Summary

4. Character Device Drivers

- The concept behind major and minor
 - Device number allocation and freeing

- Introduction to device file operations
 - File representation in the kernel

- Allocating and registering a character device

- Writing file operations
 - Exchanging data between kernel space and user space
 - A single value copy

- The open method
 - Per-device data

- The release method

- The write method
 - Steps to write

- The read method
 - Steps to read

- The llseek method
 - Steps to llseek

- The poll method
 - Steps to poll

- The ioctl method
 - Generating ioctl numbers (command)

- Steps for ioctl

- Filling the file_operations structure

- Summary

5. Platform Device Drivers

- Platform drivers

- Platform devices
 - Resources and platform data
 - Device provisioning - the old and depreciated way
 - Resources

- Platform data

Where to declare platform devices?

Device provisioning - the new and recommended way

Devices, drivers, and bus matching

How can platform devices and platform drivers match?

Kernel devices and drivers-matching function

OF style and ACPI match

ID table matching

Name matching - platform device name matching

Summary

6. The Concept of Device Tree

Device tree mechanism

Naming convention

Aliases, labels, and phandle

DT compiler

Representing and addressing devices

SPI and I2C addressing

Platform device addressing

Handling resources

Concept of named resources

Accessing registers

Handling interrupts

The interrupt handler

Interrupt controller code

Extract application-specific data

Text string

Cells and unsigned 32-bit integers

Boolean

Extract and parse sub-nodes

Platform drivers and DT

OF match style

Dealing with non-device tree platforms

Support multiple hardware with per device-specific data

Match style mixing

Platform resources and DT

Platform data versus DT

Summary

7. I2C Client Drivers

The driver architecture

The i2c_driver structure

The probe() function

Per-device data

The remove() function

Driver initialization and registration

Driver and device provisioning

Accessing the client

Plain I2C communication

System Management Bus (SMBus) compatible functions

Instantiating I2C devices in the board configuration file (old and depreciated way)

I2C and the device tree

Defining and registering the I2C driver

Remark

Instantiating I2C devices in the device tree - the new way

Putting it all together

Summary

8. SPI Device Drivers

The driver architecture

The device structure

spi_driver structure

- The probe() function
 - Per-device data

- The remove() function

- Driver initialization and registration

- Driver and devices provisioning

- Instantiate SPI devices in board configuration file – old and depreciated way

- SPI and device tree

- Instantiate SPI devices in device tree - the new way

- Define and register SPI driver

- Accessing and talking to the client

- Putting it all together

- SPI user mode driver

- With IOCTL

- Summary

9. Regmap API – A Register Map Abstraction

- Programming with the regmap API

- regmap_config structure

- regmap initialization

- SPI initialization

- I2C initialization

- Device access functions

- regmap_update_bits function

- Special regmap_multi_reg_write function

- Other device access functions

- regmap and cache

- Putting it all together

- A regmap example

- Summary

10. IIO Framework

- IIO data structures

 - iio_dev structure

 - iio_info structure

- IIO channels

 - Channel attribute naming conventions

 - Distinguishing channels

- Putting it all together

- Triggered buffer support

 - IIO trigger and sysfs (user space)

 - Sysfs trigger interface

 - add_trigger file

 - remove_trigger file

 - Tying a device with a trigger

 - The interrupt trigger interface

 - The hrtimer trigger interface

- IIO buffers

 - IIO buffer sysfs interface

 - IIO buffer setup

- Putting it all together

- IIO data access

 - One-shot capture

 - Buffer data access

 - Capturing using the sysfs trigger

 - Capturing using the hrtimer trigger

- IIO tools

- Summary

11. Kernel Memory Management

- System memory layout - kernel space and user space

 - Kernel addresses – concept of low and high memory

- Low memory
- High memory
- User space addresses
- Virtual Memory Area (VMA)
- Address translation and MMU
 - Page look up and TLB
 - How does TLB work
- Memory allocation mechanism
 - Page allocator
 - Page allocation API
 - Conversion functions
- Slab allocator
 - The buddy algorithm
 - A journey into the slab allocator
- kmalloc family allocation
- vmalloc allocator
- Process memory allocation under the hood
 - The copy-on-write (CoW) case
- Work with I/O memory to talk with hardware
 - PIO devices access
 - MMIO devices access
 - __iomem cookie
- Memory (re)mapping
 - kmap
 - Mapping kernel memory to user space
 - Using remap_pfn_range
 - Using io_remap_pfn_range
 - The mmap file operation
 - Implementing mmap in the kernel
- Linux caching system

What is a cache?

CPU cache – memory caching

The Linux page cache – disk caching

Specialized caches (user space caching)

Why delay writing data to disk?

Write caching strategies

The flusher threads

Device-managed resources – Devres

Summary

12. DMA – Direct Memory Access

Setting up DMA mappings

Cache coherency and DMA

DMA mappings

Coherent mapping

Streaming DMA mapping

Single buffer mapping

Scatter/gather mapping

Concept of completion

DMA engine API

Allocate a DMA slave channel

Set slave and controller specific parameters

Get a descriptor for transaction

Submit the transaction

Issue pending DMA requests and wait for callback notification

Putting it all together – NXP SDMA (i.MX6)

DMA DT binding

Consumer binding

Summary

13. Linux Device Model

- LDM data structures

 - The bus

 - Bus registration

 - Device driver

 - Device driver registration

 - Device

 - Device registration

- Deep inside LDM

 - kobject structure

 - kobj_type

 - ksets

 - Attribute

 - Attributes group

- Device model and sysfs

 - Sysfs files and attributes

 - Current interfaces

 - Device attributes

 - Bus attributes

 - Device drivers attributes

 - Class attributes

 - Allow sysfs attribute files to be pollable

- Summary

14. Pin Control and GPIO Subsystem

- Pin control subsystem

 - Pinctrl and the device tree

- The GPIO subsystem

 - The integer-based GPIO interface: legacy

 - Claiming and configuring the GPIO

 - Accessing the GPIO – getting/setting the value

 - In atomic context

In a non-atomic context (that may sleep)

GPIOs mapped to IRQ

Putting it all together

The descriptor-based GPIO interface: the new and recommended way
GPIO descriptor mapping - the device tree

Allocating and using GPIO

Putting it all together

The GPIO interface and the device tree
The legacy integer-based interface and device tree

GPIO mapping to IRQ in the device tree

GPIO and sysfs
Exporting a GPIO from kernel code

Summary

15. GPIO Controller Drivers – gpio_chip

Driver architecture and data structures

Pin controller guideline

Sysfs interface for GPIO controller

GPIO controllers and DT

Summary

16. Advanced IRQ Management

Multiplexing interrupts and interrupt controllers

Advanced peripheral IRQs management

Interrupt request and propagation
Chaining IRQ
Chained interrupts

Nested interrupts

Case study – GPIO and IRQ chip
Legacy GPIO and IRQ chip

New gpiolib irqchip API

Interrupt controller and DT

Summary

17. Input Devices Drivers

Input device structures

Allocating and registering an input device

Polled input device sub-class

Generating and reporting an input event

User space interface

Putting it all together

Driver examples

Summary

18. RTC Drivers

RTC framework data structures

RTC API

Reading and setting time

Driver example

Playing with alarms

RTCs and user space

The sysfs interface

The hwclock utility

Summary

19. PWM Drivers

PWM controller driver

Driver example

PWM controller binding

PWM consumer interface

PWM clients binding

Using PWMs with the sysfs interface

Summary

20. Regulator Framework

- PMIC/producer driver interface

 - Driver data structures

 - Description structure

 - Constraints structure

 - init data structure

 - Feeding init data into a board file

 - Feeding init data into the DT

 - Configuration structure

 - Device operation structure

- Driver methods

 - Probe function

 - Remove function

 - Case study: Intersil ISL6271A voltage regulator

- Driver example

- Regulators consumer interface

 - Regulator device requesting

 - Controlling the regulator device

 - Regulator output enable and disable

 - Voltage control and status

 - Current limit control and status

 - Operating mode control and status

- Regulator binding

- Summary

21. Framebuffer Drivers

- Driver data structures

- Device methods

- Driver methods
 - Detailed fb_ops
 - Checking information
 - Set controller's parameters
 - Screen blanking
 - Accelerated methods
- Putting it all together
- Framebuffer from user space
- Summary

22. Network Interface Card Drivers

- Driver data structures
 - The socket buffer structure
 - Socket buffer allocation
- Network interface structure
- The device methods
 - Opening and closing
 - Packet handling
 - Packet reception
 - Packet transmission
- Driver example
- Status and control
 - The interrupt handler
 - Ethtool support
- Driver methods
 - The probe function
 - Module unloading
- Summary

Preface

The Linux kernel is a complex, portable, modular, and widely used piece of software, running on around 80% of servers and embedded systems in more than half of the devices throughout the world. Device drivers play a critical role in the context of how well a Linux system performs. As Linux has turned out to be one of the most popular operating systems interest in developing personal device drivers is also increasing steadily.

A device driver is the link between the user space and devices, through the kernel.

This book will begins with two chapters that will help you understand the basics of drivers and prepare you for the long journey through the Linux kernel. This book will then cover driver development based on Linux subsystems such as memory management, PWM, RTC, IIO, GPIO, IRQ management. The book will also cover practical approach to direct memory access and network device drivers.

Source code in this book has been tested on both x86 PC and UDOO Quad from SECO, which is based on an ARM i.MX6 from NXP, with enough features and connections to allow us to cover all of tests discussed in the book. Some drivers are also provided for testing purposes for inexpensive components such as MCP23016 and 24LC512, which are I2C GPIO controller and eeprom memory respectively.

By the end of this book, you will be comfortable with the concept of device driver development and will be in a position to write any device driver from scratch using the latest kernel version (v4.13 at the time of writing).

What this book covers

Chapter 1, Introduction to Kernel Development, introduces the Linux kernel development process. The chapter will discuss the downloading, configuring, and compiling steps of a kernel, as well for x86 as for ARM-based systems *Chapter 2, Device Driver Basis*, deals with Linux modularity by means of kernel modules, and describes their loading/unloading. It also describes a driver architecture and some basic concepts and some kernel best practices.

Chapter 3, Kernel Facilities and Helper Functions, walks through frequently used kernel functions and mechanisms, such as work queue, wait queue, mutexes, spinlock, and any other facilities that are useful for improved driver reliability.

Chapter 4, Character Device Drivers, focuses exporting a device's functionalities to the user space by means of character devices as well as supporting custom commands using the IOCTL interface.

Chapter 5, Platform Device Drivers, explains what a platform device is and introduces the concept of pseudo-platform bus, as well as the device and bus matching mechanism. This chapter describes platform driver architecture in a general manner, and how to handle platform data.

Chapter 6, The Concept of Device Tree, discusses the mechanism to feed device descriptions to the kernel. This chapter explains device addressing, resource handling, every data type supported in DT and their kernel APIs.

Chapter 7, I2C Client Drivers, dives into I2C device drivers architecture, the data structures and device addressing and accessing methods on the bus.

Chapter 8, SPI Device Drivers, describe SPI-based device driver architecture, as well as the data structures involved. The chapter discusses each device's access method and specificities, as well as traps one should avoid. SPI DT binding is discussed too.

Chapter 9, Regmap API – A Register Map Abstraction, provides an overview of the regmap API, and how it abstracts the underlying SPI and I2C transaction. This

chapter describes the generic API, as well as the dedicated API.

Chapter 10, IIO framework, introduce the kernel data acquisition and measurement framework, to handle Digital to Analog Converters (DACs) and Analog to Digital Converters (ADCs). This walk through the IIO API, deals with triggered buffers and continuous data capture, and looks at single channel acquisition through the sysfs interface.

Chapter 11, Kernel Memory Management, first introduces the concept of virtual memory, in order to describe the whole kernel memory layout. This chapter walks through the kernel memory management subsystem, discussing memory allocation and mapping, their APIs and all devices involved in such mechanisms, as well as kernel caching mechanism.

Chapter 12, DMA – Direct Memory Access, introduce DMA and its new kernel API: the DMA Engine API. This chapter will talk about different DMA mappings and describes how to address cache coherency issues. In addition, the chapter summarize the whole concepts in use cases, based on i.MX6 SoC, from NXP.

Chapter 13, Linux Device Model, provides an overview of the heart of Linux, describing how objects are represented in the kernel, and how Linux is designed under the hood in a general manner, starting from kobject to devices, through buses, classes, and device drivers. This chapter also highlight sometime unknown side in user space, the kernel object hierarchy in sysfs.

Chapter 14, Pin Control and GPIO Subsystem, describes the kernel pincontrol API and GPIOLIB, which is the kernel API to handle GPIO. This chapter also discusses the old and deprecated integer-based GPIO interface, as well as the descriptor-based interface, which is the new one, and finally, the way they can be configured from within the DT.

Chapter 15, GPIO Controller Drivers – gpio_chip, necessary elements to write such device drivers. That says, its main data structure is struct gpio_chip. This structure is explained in detail in this chapter, along with a full and working driver provided in the source of the book.

Chapter 16, Advanced IRQ Management, demystifies the Linux IRQ core. This

chapter walks through Linux IRQ management, starting from interrupt propagation over the system and moving to interrupt controller drivers, thus explaining the concept of IRQ multiplexing, using the Linux IRQ domain API [Chapter 17](#), *Input Devices Drivers*, provides a global view of input subsystems, dealing with both IRQ-based and polled input devices, and introducing both APIs. This chapter explains and shows how user space code deals with such devices.

[Chapter 18](#), *RTC Drivers*, walks through and demystifies the RTC subsystem and its API. This chapter goes far enough and explains how to deal with alarms from within RTC drivers

[Chapter 19](#), *PWM Drivers*, provides a full description of the PWM framework, talking about the controller side API as well the consumer side API. PWM management from the user space is discussed in the last section in this chapter.

[Chapter 20](#), *Regulator Framework*, highlights how important power management is. The first part of the chapter deals with Power Management IC (PMIC) and explains its driver design and API. The second part focuses on the consumer side, talking about requesting and using regulators.

[Chapter 21](#), *Framebuffer Drivers*, explains framebuffer concept and how it works. It also shows how to design framebuffer drivers, walks through its API, and discusses accelerated as well as non-accelerated methods. This chapter shows how drivers can expose framebuffer memory so that user space can write into, without worrying about underlying tasks.

[Chapter 22](#), *Network Interface Card Drivers*, walk through the NIC driver's architecture and their data structures, thus showing you how to handle device configuration, data transfer, and socket buffers.

What you need for this book

This book assumes a medium level of understanding the Linux operating system, basic knowledge of C programming (at least pointer handling). That is all. If additional skill is required for a given chapter, links on document reference will be provided to readers to quickly learn these skills.

Linux kernel compiling is a quite long and heavy task. The minimum hardware or virtual requirements are as the follows:

- CPU: 4 cores
- Memory: 4 GB RAM
- Free disk space: 5 GB (large enough)

In this book, you will need the following software list:

- Linux operating system: preferably a Debian-based distribution, which is used for example in the book (Ubuntu 16.04)
- At least version 5 of both gcc and gcc-arm-linux (as used in the book)

Other necessary packages are described in dedicated chapter in the book. Internet connectivity is required for kernel sources downloading.

Who this book is for

To make usage of the content of this book, a basic prior knowledge of C programming and basics Linux commands is expected. This book covers Linux drivers development for widely used embedded devices, using the kernel version v4.1, and covers changes until the last version at the time of writing this book (v4.13). This book is essentially intended for embedded engineers, Linux system administrators, developer, and kernel hackers. Whether you are a software developer, a system architect, or maker willing to dive into Linux driver development, this book is for you.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `.name` field must be the same as the device's name you give when you register the device in the board specific file".

A block of code is set as follows:

```
#include <linux/of.h>
#include <linux/of_device.h>
```

Any command-line input or output is written as follows:

```
sudo apt-get update
sudo apt-get install linux-headers-$(uname -r)
```

New terms and **important words** are shown in bold.



Warnings or important notes appear like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply email feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your email address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zippeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Linux-Device-Drivers-Development>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/LinuxDeviceDriversDevelopment_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Introduction to Kernel Development

Linux started as a hobby project in 1991 for a Finnish student, Linus Torvalds. The project has gradually grown and still does, with roughly 1000 contributors around the world. Nowadays, Linux is a must, in embedded systems as well as on servers. A kernel is a center part of an operating system, and its development is not so obvious.

Linux offers many advantages over other operating systems:

- It is free of charge
- Well documented with a large community
- Portable across different platforms
- Provides access to the source code
- Lots of free open source software

This book tries to be as generic as possible. There is a special topic, device tree, which is not a full x86 feature yet. That topic will then be dedicated to ARM processors, and all those fully supporting the device tree. Why those architectures? Because they are most used on the desktop and servers (for x86) and on embedded systems (ARM).

This chapter deals among others with:

- Development environment setup
- Getting, configure, and build kernel sources
- Kernel source code organization
- Introduction to kernel coding style

Environment setup

Before one starts any development, you need to set an environment up. The environment dedicated to Linux development is quite simple, at least, on Debian based systems:

```
$ sudo apt-get update
$ sudo apt-get install gawk wget git diffstat unzip texinfo \
gcc-multilib build-essential chrpath socat libsdl1.2-dev \
xterm ncurses-dev lzop
```

There are parts of codes in this book that are compatible with ARM **system on chip (SoC)**. One should install `gcc-arm` as well:

```
| sudo apt-get install gcc-arm-linux-gnueabi
```

I'm running Ubuntu 16.04, on an ASUS RoG, with an Intel core i7 (8 physical cores), 16 GB of RAM, 256 GB of SSD, and 1 TB of magnetic hard drive. My favorite editor is Vim, but you are free to use the one you are most comfortable with.

Getting the sources

In the early kernel days (until 2003), odd–even versioning styles were used; where odd numbers were stable, even numbers were unstable. When the 2.6 version was released, the versioning scheme switched to X.Y.Z, where:

- x: This was the actual kernel's version, also called major, it incremented when there were backwards-incompatible API changes
- y: This was the minor revision, it incremented after adding a functionality in a backwards-compatible manner
- z: This is also called PATCH, represented the version relative to bug fixes

This is called semantic versioning, and has been used until the 2.6.39 version; when Linus Torvalds decided to bump the version to 3.0, which also meant the end of semantic versioning in 2011, and then, an X.Y scheme was adopted.

When it came to the 3.20 version, Linus argued that he could no longer increase Y, and decided to switch to an arbitrary versioning scheme, incrementing X whenever Y got large enough that he ran out of fingers and toes to count it. This is the reason why the version has moved from 3.20 to 4.0 directly. Have a look at: <https://plus.google.com/+LinusTorvalds/posts/jmtzzLiiejc>.

Now the kernel uses an arbitrary X.Y versioning scheme, which has nothing to do with semantic versioning.

> git clone https://github.com/torvalds/linux

git checkout v4.1

ls

- **arch/**: The Linux kernel is a fast growing project that supports more and more architectures. That being said, the kernel wants to be as generic as possible. Architecture specific code is separated from the rest, and falls in this directory. This directory contains processor-specific subdirectories such as **alpha/**, **arm/**, **mips/**, **blackfin/**, and so on.
- **block/**: This directory contains codes for block storage devices, actually the scheduling algorithm.
- **crypto/**: This directory contains the cryptographic API and the encryption algorithms code.
- **Documentation/**: This should be your favorite directory. It contains the descriptions of APIs used for different kernel frameworks and subsystems. You should look here prior to asking any questions on forums.
- **drivers/**: This is the heaviest directory, continuously growing as device drivers get merged. It contains every device driver organized in various subdirectories.
- **fs/**: This directory contains the implementation of different filesystems that the kernel actually supports, such as NTFS, FAT, ETX{2,3,4}, sysfs, procfs, NFS, and so on.
- **include/**: This contains kernel header files.
- **init/**: This directory contains the initialization and start up code.
- **ipc/**: This contains implementation of the **Inter-Process Communication (IPC)** mechanisms, such as message queues, semaphores, and shared memory.
- **kernel/**: This directory contains architecture-independent portions of the base kernel.

- `lib/`: Library routines and some helper functions live here. They are: generic **kernel object (kobject)** handlers and **Cyclic Redundancy Code (CRC)** computation functions, and so on.
- `mm/`: This contains memory management code.
- `net/`: This contains networking (whatever network type it is) protocols code.
- `scripts/`: This contains scripts and tools used during the kernel development. There are other useful tools here.
- `security/`: This directory contains the security framework code.
- `sound/`: Audio subsystems codes fall here.
- `usr/`: This currently contains the `initramfs` implementation.

The kernel must remain portable. Any architecture-specific code should be located in the `arch` directory. Of course, the kernel code related to user space API does not change (system calls, `/proc`, `/sys`), as it would break the existing programs.

The book deals with version 4.1 of the kernel. Therefore, any changes made until v4.11 version are covered too, at least this can be said about the frameworks and subsystems.

Kernel configuration

The Linux kernel is a makefile-based project, with 1000s of options and drivers. To configure your kernel, either use `make menuconfig` for an ncurses-based interface or `make xconfig` for an X-based interface. Once chosen, options will be stored in a `.config` file, at the root of the source tree.

In most of the cases, there will be no need to start a configuration from scratch. There are default and useful configuration files available in each `arch` directory, which you can use as a start point:

```
| ls arch/<you_arch>/configs/
```

For ARM-based CPUs, these configs files are located in `arch/arm/configs/`, and for an i.MX6 processor, the default file config is `arch/arm/configs/imx_v6_v7_defconfig`. Similarly for x86 processors, we find the files in `arch/x86/configs/`, with only two default configuration files, `i386_defconfig` and `x86_64_defconfig`, for 32 and 64 bits versions respectively. It is quite straightforward for an x86 system:

```
| make x86_64_defconfig
| make zImage -j16
| make modules
| make INSTALL_MOD_PATH </where/to/install> modules_install
```

Given an i.MX6-based board, one can start with `ARCH=arm make imx_v6_v7_defconfig`, and then `ARCH=arm make menuconfig`. With the former command, you will store the default option in `.config` file, and with the latter, you can update add/remove options, depending on the needs.

One may run into a Qt4 error with `xconfig`. In such a case, one should just use the following command:

```
| sudo apt-get install qt4-dev-tools qt4-qmake
```


Build your kernel

Building the kernel requires you to specify the architecture for which it is built for, as well as the compiler. That says, it is not necessary for a native build.

```
| ARCH=arm make imx_v6_v7_defconfig  
| ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make zImage -j16
```

After that, one will see something like:

```
| [...]
| LZO      arch/arm/boot/compressed/piggy_data
| CC       arch/arm/boot/compressed/misc.o
| CC       arch/arm/boot/compressed/decompress.o
| CC       arch/arm/boot/compressed/string.o
| SHIPPED  arch/arm/boot/compressed/hyp-stub.S
| SHIPPED  arch/arm/boot/compressed/lib1funcs.S
| SHIPPED  arch/arm/boot/compressed/ashldi3.S
| SHIPPED  arch/arm/boot/compressed/bswapsdi2.S
| AS       arch/arm/boot/compressed/hyp-stub.o
| AS       arch/arm/boot/compressed/lib1funcs.o
| AS       arch/arm/boot/compressed/ashldi3.o
| AS       arch/arm/boot/compressed/bswapsdi2.o
| AS       arch/arm/boot/compressed/piggy.o
| LD       arch/arm/boot/compressed/vmlinux
| OBJCOPY  arch/arm/boot/zImage
| Kernel: arch/arm/boot/zImage is ready
```

From the kernel build, the result will be a single binary image, located in `arch/arm/boot/`. Modules are built with the following command:

```
| ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make modules
```

You can install them using the following command:

```
| ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make modules_install
```

The `modules_install` target expects an environment variable, `INSTALL_MOD_PATH`, which specifies where you should install the modules. If not set, the modules will be installed at `/lib/modules/$(KERNELRELEASE)/kernel/`. This is discussed in [Chapter 2, Device Driver Basis](#).

i.MX6 processors support device trees, which are files you use to describe the hardware (this is discussed in detail in [Chapter 6, The Concept of Device Tree](#)). However, to compile every `ARCH` device tree, you can run the following command:

```
| ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make dtbs
```

However, the `dtbs` option is not available on all platforms that support device tree. To build a standalone DTB, you should use:

```
| ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make imx6d-      sabrelite.dtb
```

Kernel habits

The kernel code tries to follow standard rules through its evolution. In this chapter, we will just be introduced to them. They are all discussed in a dedicated chapter, starting from [Chapter 3](#), *Kernel Facilities and Helper Functions*, we get a better overview of the kernel development process and tips, till [Chapter 13](#), *Linux Device Model*.

**sudo apt-get install indent scripts/Lindent
my_module.c**

- Every function/variable that is not exported should be declared as static.
- No spaces should be added around (inside) parenthesized expressions. *s = size of (struct file);* is accepted, whereas *s = size of(struct file);* is not.
- Using typedefs is forbidden.
- Always use `/* this */` comment style, not `// this`
- - BAD: `// do not use this please`
 - GOOD: `/* Kernel developers like this */`
- You should capitalise macros, but functional macros can be in lowercase.
- A comment should not replace a code that is not illegible. Prefer rewriting the code rather than adding a comment.

Kernel structures allocation/initialization

The kernel always offers two possible allocation mechanisms for its data structures and facilities.

Some of these structures are:

- Workqueue
- List
- Waitqueue
- Tasklet
- Timer
- Completion
- mutex
- spinlock

Dynamical initializers are all macros it means they are always capitalized:

`INIT_LIST_HEAD()`, `DECLARE_WAIT_QUEUE_HEAD()`, `DECLARE_TASKLET()`, and so on.

That being said, these are all discussed in [Chapter 3](#), *Kernel Facilities and Helper Functions*. Therefore, data structures that represent framework devices are always allocated dynamically, each of which having its own allocation and deallocation API. These framework device types are:

- Network
- Input device
- Char device
- IIO device
- Class
- Framebuffer
- Regulator
- PWM device
- RTC

Scope of the static objects is visible in the whole driver, and by every device this driver manages. Dynamically allocated objects are visible only by the device that is actually using a given instance of the module.

Classes, objects, and OOP

The kernel implements OOP by means of a device and a class. Kernel subsystems are abstracted by means of classes. There are almost as many subsystems as there are directories under `/sys/class/`. The `struct kobject` structure is the center piece of this implementation. It even brings in a reference counter, so that the kernel may know how many users actually use the object. Every object has a parent, and has an entry in `sysfs` (if mounted).

Every device that falls into a given subsystem has a pointer to an **operations (ops)** structure, which exposes operations that can be executed on this device.

Summary

This chapter explained in a very short and simple manner how you should download the Linux source and process a first build. It also deals with some common concepts. That said, this chapter is quite brief and may not be enough, but never mind, it is just an introduction. That is why the next chapter gets more into the details of the kernel building process, how to actually compile a driver, either externally or as a part of the kernel, as well as some basics that one should learn before starting the long journey that kernel development represents.

Device Driver Basis

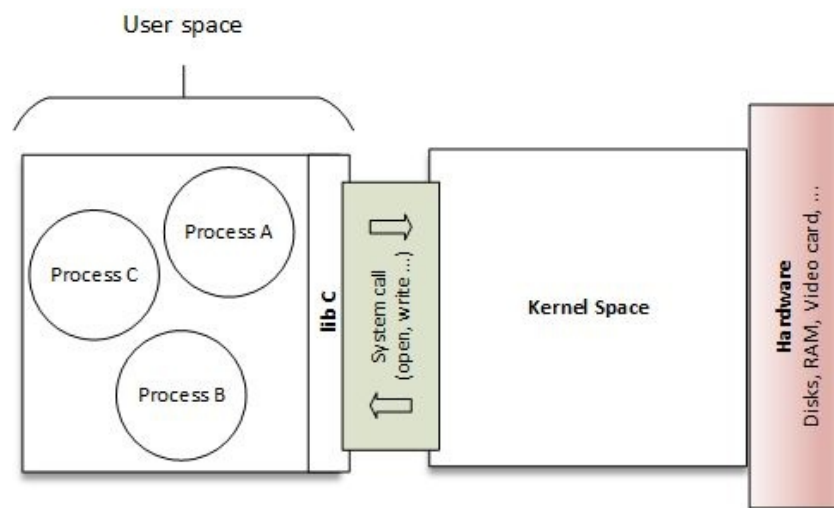
A driver is a piece of software whose aim is to control and manage a particular hardware device; hence the name device driver. From an operating system point of view, it can be either in the kernel space (running in privileged mode) or in the user space (with lower privilege). This book only deals with kernel space drivers, especially Linux kernel drivers. Our definition is a device driver exposes the functionality of the hardware to user programs.

This book's aim is not to teach you how to become a Linux guru—I'm not even one at all—but there are some concepts you should understand prior to writing a device driver. C programming skills are mandatory; you should be at least familiar with pointers. You should also be familiar with some of the manipulating functions. Some hardware skills are required too. So this chapter essentially discusses:

- Module building processes, as well as their loading and unloading
- Driver skeletons, and debugging message management
- Error handling in the driver

User space and kernel space

The concept of kernel space and user space is a bit abstract. It is all about memory and access rights. One may consider the kernel to be privileged, whereas the user apps are restricted. It is a feature of a modern CPU, allowing it to operate either in privileged or unprivileged mode. This concept will be clearer to you in [Chapter 11, Kernel Memory Management](#).



User space and kernel space

The preceding figure introduces the separation between kernel and user space, and highlights the fact that system calls represent the bridge between them (we discuss this later in this chapter). One can describe each space as following:

- **Kernel space:** This is a set of addresses where the kernel is hosted and where it runs. Kernel memory (or kernel space) is a memory range, owned by the kernel, protected by access flags, preventing any user apps from messing with the kernel (un)knowingly. On the other hand the kernel can access the whole system memory, since it runs with the higher priority on the system. In kernel mode, the CPU can access the whole memory (both kernel space and user space).
- **User space:** This is a set of addresses (locations) where normal programs (such as gedit and so on) are restricted to run. You may consider it as a

sand-box or a jail, so that a user program can't mess with memory or any other resource owned by another program. In user mode, the CPU can only access memory tagged with user-space access rights. The only way for the user app to run into the kernel space is through system calls. Some of these are `read`, `write`, `open`, `close`, `mmap`, and so on. User-space code runs with lower priority. When a process performs a system call, a software interrupt is sent to the kernel, which turns on privileged mode so that the process can run in kernel space. When the system call returns, the kernel turns off the privileged mode and the process is jailed again.

CONFIG_MODULES=y

Module dependencies

In Linux, a module can provide functions or variables, exporting them using the `EXPORT_SYMBOL` macro, which makes them available for other modules. These are called symbols. A dependency of module B on module A is that module B is using one of the symbols exported by module A.

depmod utility

depmod is a tool that you run during the kernel build process to generate module dependency files. It does that by reading each module in `/lib/modules/<kernel_release>/` to determine what symbols it should export and what symbols it needs. The result of that process is written to the file `modules.dep`, and its binary version `modules.dep.bin`. It is a kind of module indexing.

Module loading and unloading

For a module to be operational, one should load it into the kernel, either by using `insmod` given the module path as argument, which is the preferred method during development, or by using `modprobe`, a clever command but that one preferred in production systems.

Manual loading

Manual loading needs the intervention of a user, which should have root access. The two classical methods to achieve this are described as follows:

modprobe and insmod

During development, one usually uses `insmod` in order to load a module and it should be given the path of the module to load: **`insmod /path/to/mydrv.ko`**

It is low-level form of module loading, which forms the base of other module loading methods, and the one we will use in this book. On the other hand, there is `modprobe`, mostly used by `sysadmin` or in a production system. `modprobe` is a clever command that parses the file `modules.dep` in order to load dependencies first, prior to loading the given module. It automatically handles module dependencies, as a package manager does: **`modprobe mydrv`**

Whether one can use `modprobe` or not depends on `depmod` being aware of module installation.

#this line is a comment

uio

iwlwifi

Auto-loading

The `depmod` utility doesn't only build `modules.dep` and `modules.dep.bin` files. It does more than that. When a kernel developer actually writes a driver, they know exactly what hardware the driver will support. They are then responsible for feeding the driver with the product and vendor IDs of all devices supported by the driver. `depmod` also processes module files in order to extract and gather that information, and generates a `modules.alias` file, located in

`/lib/modules/<kernel_release>/modules.alias`, which will map devices to their drivers:

An excerpt of `modules.alias` is as follows: `alias`

```
usb:v0403pFF1Cd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio alias
usb:v0403pFF18d*dc*dsc*dp*ic*isc*ip*in* ftdi_sio alias
usb:v0403pDAFFd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio alias
usb:v0403pDAFE d*dc*dsc*dp*ic*isc*ip*in* ftdi_sio alias
usb:v0403pDAFDd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio alias
usb:v0403pDAFCd*dc*dsc*dp*ic*isc*ip*in* ftdi_sio alias
usb:v0D8Cp0103d*dc*dsc*dp*ic*isc*ip*in* snd_usb_audio alias
usb:v*p*d*dc*dsc*dp*ic01isc03ip*in* snd_usb_audio alias
usb:v200Cp100Bd*dc*dsc*dp*ic*isc*ip*in* snd_usb_au
```

At this step, you'll need a user-space hot-plug agent (or device manager), usually `udev` (or `mdev`), that will register with the kernel in order to get notified when a new device appears.

The notification is done by the kernel, sending the device's description (`pid`, `vid`, `class`, `device class`, `device subclass`, `interface`, and all other information that may identify a device) to the hot-plug daemon, which in turn calls `modprobe` with this information. `modprobe` then parses the `modules.alias` file in order to match the driver associated with the device. Before loading the module, `modprobe` will look for its dependencies in `module.dep`. If it finds any, the dependencies will be loaded prior to the associated module loading; otherwise, the module is loaded directly.

Module unload

The usual command to unload a module is `rmmod`. One should prefer using this to unload a module loaded with `insmod` command. The command should be given the module name to unload as a parameter. Module unloading is a kernel feature that one can enable or disable, according to the value of the `CONFIG_MODULE_UNLOAD` config option. Without this option, one will not be able to unload any module. Let us enable module unloading support:

```
| CONFIG_MODULE_UNLOAD=y
```

At runtime, the kernel will prevent from unloading modules that may break things, even if one asks it to do so. This is because the kernel keeps a reference count on module usage, so that it knows whether a module is actually in use or not. If the kernel believes it is unsafe to remove a module, it will not. Obviously, one can change this behavior:

```
| MODULE_FORCE_UNLOAD=y
```

The preceding option should be set in the kernel config in order to force module unload:

```
| rmmod -f mymodule
```

On the other hand, a higher level command to unload a module in a smart manner is `modprobe -r`, which automatically unloads unused dependencies:

```
| modprobe -r mymodule
```

As you may have guessed, it is a really helpful option for developers. Finally, one can check whether a module is loaded or not with the following command:

```
| lsmod
```

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

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

```
static int __init helloworld_init(void) {
```

```
    pr_info("Hello world!\n"); return 0; }
```

```
static void __exit helloworld_exit(void) {
```

```
    pr_info("End of the world\n"); }
```

```
module_init(helloworld_init);
```

```
module_exit(helloworld_exit);
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
```

```
MODULE_LICENSE("GPL");
```

Module entry and exit point

Kernel drivers all have entry and exit points: the former corresponds to the function called when the module is loaded (`modprobe`, `insmod`) and the latter is the function executed at module unloading (at `rmmod` or `modprobe -r`).

We all remember the `main()` function, which is the entry point for every user-space program written in C/C++ that exits when that same function returns. With kernel modules, things are different. The entry point can have any name you want, and unlike a user-space program that exits when `main()` returns, the exit point is defined in another function. All you need to do is to inform the kernel which functions should be executed as an entry or exit point. The actual functions `helloworld_init` and `helloworld_exit` could be given any name. The only thing that is actually mandatory is to identify them as the corresponding loading and removing functions, giving them as parameters to the `module_init()` and `module_exit()` macros.

To sum up, `module_init()` is used to declare the function that should be called when the module is loaded (with `insmod` or `modprobe`). What is done in the initialization function will define the behavior of the module. `module_exit()` is used to declare the function that should be called when the module is unloaded (with `rmmod`).



Either the `init` function or the `exit` function is run once, right after the module is loaded or unloaded.

`__init` and `__exit` attributes

`__init` and `__exit` are actually kernel macros, defined in `include/linux/init.h`, shown as follows:

```
| #define __init__section(.init.text)  
| #define __exit__section(.exit.text)
```

The `__init` keyword tells the linker to place the code in a dedicated section into the kernel object file. This section is known in advance to the kernel, and freed when the module is loaded and the `init` function finished. This applies only to built-in drivers, not to loadable modules. The kernel will run the `init` function of the driver for the first time during its boot sequence.

Since the driver cannot be unloaded, its `init` function will not be called again until the next reboot. There is no need to keep references on its `init` function anymore. The same for the `__exit` keyword, whose corresponding code is omitted when the module is compiled statically into the kernel, or when module unloading support is not enabled, because in both cases, the `exit` function is never called. `__exit` has no effect on loadable modules.

Let's spend more time understanding how such attributes work. It is all about object files called **Executable and Linkable Format (ELF)**. An ELF object file is made of various named sections. Some of these are mandatory and form the basis of the ELF standard, but one can make up any section one wants and have it used by special programs. This is what the kernel does. One can run `objdump -h module.ko` in order to print out different sections that constitute the given `module.ko` kernel module:

```

ma@jma:~/work/tutos/sources/helloworld$ objdump -h helloworld-params.ko
helloworld-params.ko:      file format elf64-x86-64

Sections:
Idx Name          Size      VMA           LMA             File off  Algn
 0 .note.gnu.build-id 00000024 0000000000000000 0000000000000000 00000040 2**2
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 1 .text           00000017 0000000000000000 0000000000000000 00000070 2**4
   CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 2 .init.text      00000014 0000000000000000 0000000000000000 00000087 2**0
   CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 3 .text.unlikely  00000000 0000000000000000 0000000000000000 0000009b 2**0
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 4 .rodata.str1.1  0000002a 0000000000000000 0000000000000000 0000009b 2**0
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 5 .modinfo        00000157 0000000000000000 0000000000000000 000000c8 2**3
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 __param         00000078 0000000000000000 0000000000000000 00000220 2**3
   CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
 7 .rodata         0000004c 0000000000000000 0000000000000000 000002a0 2**5
   CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
 8 __mcount_loc    00000008 0000000000000000 0000000000000000 000002f0 2**3
   CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
 9 __versions      00000180 0000000000000000 0000000000000000 00000300 2**5
   CONTENTS, ALLOC, LOAD, READONLY, DATA
10 .data           00000018 0000000000000000 0000000000000000 00000480 2**3
   CONTENTS, ALLOC, LOAD, RELOC, DATA
11 .gnu.linkonce.this module 00000380 0000000000000000 0000000000000000 000004c0 2**6
   CONTENTS, ALLOC, LOAD, RELOC, DATA, LINK ONCE DISCARD
12 .bss            00000000 0000000000000000 0000000000000000 00000840 2**0
   ALLOC
13 .comment        0000006a 0000000000000000 0000000000000000 00000840 2**0
   CONTENTS, READONLY
14 .note.GNU-stack 00000000 0000000000000000 0000000000000000 000008aa 2**0
   CONTENTS, READONLY

```

List of sections of helloworld-params.ko module

Only a few of the sections in the caption are standard ELF sections:

- `.text`, also called code, which contains program code
- `.data`, which contains initialized data, and is also called data segment
- `.rodata`, for read-only data
- `.comment`
- Uninitialized data segment, also called **block started by symbol (bss)**

Other sections are added on demand for the kernel purpose. The most important for this chapter are **.modinfo** sections, which store information about the modules, and **.init.text** sections, which store code prefixed with the `__init` macro.

The linker (`ld` on Linux systems), which is a part of `binutils`, is responsible for the placement of symbols (data, code, and so on) in the appropriate section in the generated binary in order to be processed by the loader when the program is executed. One may customize these sections, change their default location, or even add additional sections by providing a linker script, called a **linker definition file (LDF)** or **linker definition script (LDS)**. Now all you have to do is to inform the linker of the symbol placement through compiler directives. The

GNU C compiler provides attributes for that purpose. In the case of the Linux kernel, there is a custom LDS file provided, located in `arch/<arch>/kernel/vmlinux.lds.S`. `__init` and `__exit` are then used to mark symbols to be placed onto dedicated sections mapped in kernel's LDS files.

In conclusion, `__init` and `__exit` are Linux directives (actually macros), which wrap the C compiler attribute used for symbol placement. They instruct the compiler to put the code they prefix respectively in `.init.text` and `.exit.text` sections, even though the kernel can access different object sections.

Module information

Even without having to read its code, one should be able to gather some information (for example, the author(s), parameter(s) description, the license) about a given module. A kernel module uses its `.modinfo` section to store informations about the module. Any `MODULE_*` macro will update the content of that section with the values passed as parameters. Some of these macros are `MODULE_DESCRIPTION()`, `MODULE_AUTHOR()`, and `MODULE_LICENSE()`. The real underlying macro provided by the kernel to add an entry in the module info section is `MODULE_INFO(tag, info)`, which adds generic info of form `tag = info`. This means a driver author could add any free form info they want, such as:
`MODULE_INFO(my_field_name, "What easy value");`

One can dump the content of the `.modinfo` section of a kernel module using the `objdump -d -j .modinfo` command on the given module:

```

jma@jma:~/work/tutos/sources/helloworld$ objdump helloworld-params.ko -d -j .modinfo
helloworld-params.ko:      file format elf64-x86-64

Disassembly of section .modinfo:

0000000000000000 < _UNIQUE_ID_license7>:
   0:  6c 69 63 65 6e 73 65 3d 47 50 4c 00          license=GPL.

000000000000000c < _UNIQUE_ID_author6>:
   c:  61 75 74 68 6f 72 3d 4a 6f 68 6e 20 4d 61 64 69   author=John Madi
  1c:  65 75 20 3c 6a 6f 68 6e 2e 6d 61 64 69 65 75 40   eu <john.madieu@
  2c:  66 6f 6f 62 61 72 2e 63 6f 6d 3e 00          foobar.com>.

0000000000000038 < _UNIQUE_ID_myarr5>:
  38:  70 61 72 6d 3d 6d 79 61 72 72 3a 74 68 69 73 20   parm=myarr:this
  48:  69 73 20 6d 79 20 61 72 72 61 79 20 6f 66 20 69   is my array of i
  58:  6e 74 00                                         nt.

000000000000005b < _UNIQUE_ID_mystr4>:
  5b:  70 61 72 6d 3d 6d 79 73 74 72 3a 74 68 69 73 20   parm=mystr:this
  6b:  69 73 20 6d 79 20 63 68 61 72 20 70 6f 69 6e 74   is my char point
  7b:  65 72 20 76 61 72 69 61 62 6c 65 00          er variable.

0000000000000087 < _UNIQUE_ID_myint3>:
  87:  70 61 72 6d 3d 6d 79 69 6e 74 3a 74 68 69 73 20   parm=myint:this
  97:  69 73 20 6d 79 20 69 6e 74 20 76 61 72 69 61 62   is my int variab
 a7:  6c 65 00                                         le.

00000000000000aa < _UNIQUE_ID_myarrtype2>:
 aa:  70 61 72 6d 74 79 70 65 3d 6d 79 61 72 72 3a 61   parmtype=myarr:a
 ba:  72 72 61 79 20 6f 66 20 69 6e 74 00          rray of int.

00000000000000c6 < _UNIQUE_ID_mystrtype1>:
 c6:  70 61 72 6d 74 79 70 65 3d 6d 79 73 74 72 3a 63   parmtype=mystr:c
 d6:  68 61 72 70 00                                harp.

00000000000000db < _UNIQUE_ID_myinttype0>:
 db:  70 61 72 6d 74 79 70 65 3d 6d 79 69 6e 74 3a 69   parmtype=myint:i
 eb:  6e 74 00 00 00                                nt...

00000000000000f0 < _UNIQUE_ID_srcversion1>:
 f0:  73 72 63 76 65 72 73 69 6f 6e 3d 42 42 46 34 33   srcversion=BBF43
100:  45 30 39 38 45 41 42 35 44 32 45 32 44 44 37 38   E098EAB5D2E2DD78
110:  43 30 00 00 00 00 00 00                                C0.....

0000000000000118 < _module_depends>:
118:  64 65 70 65 6e 64 73 3d 00                      depends=.

```

Content of .modinfo section of helloworld-params.ko module

The modinfo section can be seen as the data sheet of the module. The user-space tool that actually prints information in a stylized manner is `modinfo`:

```

jma@jma:~/work/tutos/sources/helloworld$ modinfo ./helloworld-params.ko
filename:      /home/jma/work/tutos/sources/helloworld/./helloworld-params.ko
license:      GPL
author:       John Madieu <john.madieu@foobar.com>
my_field_name: What eeasy value
srcversion:   47B038B61944D8CD2E680DB
depends:
vermagic:    4.4.0-93-generic SMP mod unload modversions
parm:        myint:this is my int variable (int)
parm:        mystr:this is my char pointer variable (charp)
parm:        myarr:this is my array of int (array of int)

```

modinfo output

Apart from the custom info one defines, there is standard info one should provide, and that the kernel provides macros for; these are license, module author, parameter description, module version, and module description.

MODULE_LICENSE ("GPL");

/*

* The following license idents are currently accepted as indicating
free * software modules

*

* "GPL" [GNU Public License v2 or later]

* "GPL v2" [GNU Public License v2]

* "GPL and additional rights" [GNU Public License v2 rights and
more]

* "Dual BSD/GPL" [GNU Public License v2

* or BSD license choice]

* "Dual MIT/GPL" [GNU Public License v2

* or MIT license choice]

* "Dual MPL/GPL" [GNU Public License v2

* or Mozilla license choice]

*

* The following other idents are available *

* "Proprietary" [Non free products]

*

* There are dual licensed components, but when running with Linux it is the * GPL that is relevant so this is a non issue. Similarly LGPL linked with GPL

* is a GPL combined work.

*

* This exists for several reasons * 1. So modinfo can show license info for users wanting to vet their setup * is free

* 2. So the community can ignore bug reports including proprietary modules * 3. So vendors can do likewise based on their own policies */

It is mandatory for your module to be at least GPL-compatible in order for you to enjoy full kernel services.

Module author(s)

`MODULE_AUTHOR()` declares the module's author(s):

```
| MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
```

It is possible to have more than one author. In this case, each author must be declared with `MODULE_AUTHOR()`:

```
| MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");  
| MODULE_AUTHOR("Lorem Ipsum <l.ipsum@foobar.com>");
```

Module description

`MODULE_DESCRIPTION()` briefly describes what the module does:

```
| MODULE_DESCRIPTION("Hello, world! Module");
```

Errors and message printing

Error codes are interpreted either by the kernel or by the user-space application (through the `errno` variable). Error handling is very important in software development, more than it is in kernel development. Fortunately, the kernel provides a couple of errors that cover almost every error you'll encounter, and sometimes you will need to print them out in order to help you debug.

```
#define EPERM 1 /* Operation not permitted */  
  
#define ENOENT 2 /* No such file or directory */  
  
#define ESRCH 3 /* No such process */  
  
#define EINTR 4 /* Interrupted system call */  
  
#define EIO 5 /* I/O error */  
  
#define ENXIO 6 /* No such device or address */  
  
#define E2BIG 7 /* Argument list too long */  
  
#define ENOEXEC 8 /* Exec format error */  
  
#define EBADF 9 /* Bad file number */  
  
#define ECHILD 10 /* No child processes */  
  
#define EAGAIN 11 /* Try again */  
  
#define ENOMEM 12 /* Out of memory */  
  
#define EACCES 13 /* Permission denied */  
  
#define EFAULT 14 /* Bad address */  
  
#define ENOTBLK 15 /* Block device required */  
  
#define EBUSY 16 /* Device or resource busy */  
  
#define EEXIST 17 /* File exists */  
  
#define EXDEV 18 /* Cross-device link */
```

```
#define ENODEV 19 /* No such device */

#define ENOTDIR 20 /* Not a directory */

#define EISDIR 21 /* Is a directory */

#define EINVAL 22 /* Invalid argument */

#define ENFILE 23 /* File table overflow */

#define EMFILE 24 /* Too many open files */

#define ENOTTY 25 /* Not a typewriter */

#define ETXTBSY 26 /* Text file busy */

#define EFBIG 27 /* File too large */

#define ENOSPC 28 /* No space left on device */

#define ESPIPE 29 /* Illegal seek */

#define EROFS 30 /* Read-only file system */

#define EMLINK 31 /* Too many links */

#define EPIPE 32 /* Broken pipe */

#define EDOM 33 /* Math argument out of domain of func */

#define ERANGE 34 /* Math result not representable */

dev = init(&ptr);

if(!dev)
```

```
return -EIO
```

```
#include <errno.h> /* to access errno global variable */
```

```
#include <string.h>
```

```
[...]
```

```
if(write(fd, buf, 1) < 0) {
```

```
    printf("something gone wrong! %s\n", strerror(errno)); }
```

```
[...]
```

```
ptr = kmalloc(sizeof (device_t)); if(!ptr) {
```

```
    ret = -ENOMEM
```

```
    goto err_alloc; }
```

```
dev = init(&ptr);
```

```
if(dev) {
```

```
    ret = -EIO
```

```
    goto err_init; }
```

```
return 0;
```

```
err_init:
```

```
    free(ptr); err_alloc:

    return ret;

if (ops1() != ERR) {
    if (ops2() != ERR) {
        if ( ops3() != ERR) {
            if (ops4() != ERR) {
if (ops1() == ERR) // |
    goto error1; // |
if (ops2() == ERR) // |
    goto error2; // |
if (ops3() == ERR) // |
    goto error3; // |
if (ops4() == ERR) // V
    goto error4; error5:

[...]
```

error4:

[...]

error3:

[...]

error2:

[...]

error1:

[...]

This means, one should only use goto to move forward in a function.


```

void *ERR_PTR(long error);

long IS_ERR(const void *ptr); long PTR_ERR(const void *ptr);

static struct iio_dev *iio_dev_setup(){

    [...]

    struct iio_dev *iio_dev; iio_dev =
    devm_iio_device_alloc(&data->client->dev, sizeof(data)); if
    (!iio_dev) return ERR_PTR(-ENOMEM); [...]

    return iio_dev; }

static int foo_probe([...]){

    [...]

    struct iio_dev *my_iio_dev = iio_dev_setup(); if
    (IS_ERR(my_iio_dev)) return PTR_ERR(data->acc_iio_dev);
    [...]

}

```

This is a plus on error handling, which is also an excerpt of the kernel coding style that says: If the name of a function is an action or an imperative command, the function should return an error-code integer. If the name is a predicate, the function should return a succeeded Boolean. For example, `add_work` is a command, and the `add_work()` function returns 0 for success or `-EBUSY` for failure. In the same way, `PCI device present` is a predicate, and the `pci_dev_present()` function returns 1 if it succeeds in finding a matching device or 0 if it

doesn't.

Message printing – printk()

The `printk()` is to the kernel what `printf()` is to the user-space. Lines written by `printk()` can be displayed through the `dmesg` command. Depending on how important the message you need to print is, you can choose between eight log-level messages, defined in `include/linux/kern_levels.h`, along with their meaning:

The following is the list of kernel log levels. Each of these levels correspond to a number in a string, whose priority is inverted proportional to the value of the number. For example, 0 is higher priority:

```
#define KERN_SOH      "\001"          /* ASCII Start Of Header */
#define KERN_SOH_ASCII '\001'

#define KERN_EMERG    KERN_SOH "0"    /* system is unusable */
#define KERN_ALERT    KERN_SOH "1"    /* action must be taken immediately */
#define KERN_CRIT     KERN_SOH "2"    /* critical conditions */
#define KERN_ERR      KERN_SOH "3"    /* error conditions */
#define KERN_WARNING  KERN_SOH "4"    /* warning conditions */
#define KERN_NOTICE   KERN_SOH "5"    /* normal but significant condition */
#define KERN_INFO     KERN_SOH "6"    /* informational */
#define KERN_DEBUG    KERN_SOH "7"    /* debug-level messages */
```

The following code shows how one can print a kernel message along with a log level:

```
| printk(KERN_ERR "This is an error\n");
```

If you omit the debug level (`printk("This is an error\n")`), the kernel will provide one to the function, depending on the `CONFIG_DEFAULT_MESSAGE_LOGLEVEL` config option, which is the default kernel log level. One may actually use one of the following, much more meaningful macros, which are wrappers around those defined previously: `pr_emerg`, `pr_alert`, `pr_crit`, `pr_err`, `pr_warning`, `pr_notice`, `pr_info`, and `pr_debug`:

```
| pr_err("This is the same error\n");
```

For new drivers, it is recommended to use these wrappers. The reality of `printk()` is that, whenever it is called, the kernel compares the message log level with the current console log level; if the former is higher (lower value) than the latter, the message will be immediately printed to the console. You can check your log

level parameters with:

```
| cat /proc/sys/kernel/printk  
| 4 4 1 7
```

In this code, the first value is the current log level (4), and the second is the default one, according to the `CONFIG_DEFAULT_MESSAGE_LOGLEVEL` option. Other values are not relevant for the purpose of this chapter, so let us ignore these.

A list of kernel log levels is as follows:

```
| /* integer equivalents of KERN_<LEVEL> */  
| #define LOGLEVEL_SCHED      -2    /* Deferred messages from sched code  
|                               * are set to this special level */  
| #define LOGLEVEL_DEFAULT   -1    /* default (or last) loglevel */  
| #define LOGLEVEL_EMERG      0     /* system is unusable */  
| #define LOGLEVEL_ALERT      1     /* action must be taken immediately */  
| #define LOGLEVEL_CRIT       2     /* critical conditions */  
| #define LOGLEVEL_ERR        3     /* error conditions */  
| #define LOGLEVEL_WARNING    4     /* warning conditions */  
| #define LOGLEVEL_NOTICE     5     /* normal but significant condition */  
| #define LOGLEVEL_INFO       6     /* informational */  
| #define LOGLEVEL_DEBUG      7     /* debug-level messages */
```

The current log level can be changed with:

```
| # echo <level> > /proc/sys/kernel/printk
```



printk() never blocks and is safe enough to be called even from atomic contexts. It tries to lock the console and print the message. If locking fails, the output will be written into a buffer and the function will return, never blocking. The current console holder will then be notified about new messages and will print them before releasing the console.

The kernel supports other debug methods too, either dynamically or by using `#define DEBUG` on top of the file. People interested in such debugging style can refer to kernel documentation in *Documentation/dynamic-debug-howto.txt* file.

```
module_param(name, type, perm);
```

```
#include <linux/moduleparam.h> [...]
```

```
static char *mystr = "hello"; static int myint = 1;
```

```
static int myarr[3] = {0, 1, 2};
```

```
module_param(myint, int, S_IRUGO); module_param(mystr, charp,  
S_IRUGO); module_param_array(myarr, int, NULL,  
S_IWUSR|S_IRUSR); /* */
```

```
MODULE_PARM_DESC(myint, "this is my int variable");  
MODULE_PARM_DESC(mystr, "this is my char pointer variable");  
MODULE_PARM_DESC(myarr, "this is my array of int");
```

```
static int foo()
```

```
{
```

```
    pr_info("mystring is a string: %s\n", mystr); pr_info("Array  
elements: %d\t%d\t%d", myarr[0], myarr[1], myarr[2]); return myint;  
}
```

```
<strong># insmod hellomodule-params.ko mystring="packtpub"  
myint=15 myArray=1,2,3</strong>
```

```
<strong>$ modinfo ./helloworld-params.ko</strong><br/>
```

```
<strong>filename:
```

```
/home/jma/work/tutos/sources/helloworld/./helloworld-
```

params.ko
license: GPL

author: John Madieu <john.madieu@gmail.com>

srcversion: BBF43E098EAB5D2E2DD78C0

depends:
vermagic: 4.4.0-93-
generic SMP mod_unload modversions
parm:
myint:this is my int variable (int)
parm:
mystr:this is my char pointer variable (charp)

parm: myarr:this is my array of int (array of int)

Building your first module

There are two places to build a module. It depends on whether you want people to enable the module by themselves or not using the kernel config interface.

The module's makefile

A makefile is a special file used to execute a set of actions, among which the most important is the compilation of programs. There is a dedicated tool to parse makefiles, called `make`. Prior to jumping to the description of the whole make file, let us introduce the `obj-<X>` kbuild variable.

In almost every kernel makefile, one will see at least one instance of an `obj-<X>` variable. This actually corresponds to the `obj-<X>` pattern, where `<X>` should be either `y`, `m`, left blank, or `n`. This is used by the kernel makefile from the head of the kernel build system in a general manner. These lines define the files to be built, any special compilation options, and any subdirectories to be entered recursively. A simple example is:

```
| obj-y += mymodule.o
```

This tells kbuild that there is one object in the current directory named `mymodule.o`. `mymodule.o` will be built from `mymodule.c` or `mymodule.s`. How and if `mymodule.o` will be built or linked depends on the value of `<X>`:

- If `<X>` is set to `m`, the variable `obj-m` is used, and `mymodule.o` will be built as a module.
- If `<X>` is set to `y`, the variable `obj-y` is used, and `mymodule.o` will be built as part of the kernel. One then says `foo` is a built-in module.
- If `<X>` is set to `n`, the variable `obj-m` is used, and `mymodule.o` will not be built at all.

Therefore, the pattern `obj-$(CONFIG_XXX)` is often used, where `CONFIG_XXX` is a kernel config option, set or not during the kernel configuration process. An example is:

```
| obj-$(CONFIG_MYMODULE) += mymodule.o
```

`$(CONFIG_MYMODULE)` evaluates to either `y` or `m` according to its value during the kernel configuration (remember `make menuconfig`). If `CONFIG_MYMODULE` is neither `y` nor `m`, then the file will not be compiled nor linked. `y` means built-in (it stands for yes in the kernel config process), and `m` stands for module. `$(CONFIG_MYMODULE)` pulls the right answer from the normal config process. This is explained in the next section.

The last use case is:

```
| obj-<X> += somedir/
```

This means that kbuild should go into the directory named `somedir`; look for any makefile inside and process it in order to decide what objects should be built.

Back to the makefile, the following is the content makefile we will use to build each of the modules introduced in the book:

```
obj-m := helloworld.o

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

all default: modules
install: modules_install

modules modules_install help clean:
$(MAKE) -C $(KERNELDIR) M=$(shell pwd) $@
```

- `obj-m := helloworld.o`: `obj-m` lists modules we want to build. For each `<filename>.o`, the build system will look for a `<filename>.c` to build. `obj-m` is used to build a module, whereas `obj-y` will result in a built-in object.
- `KERNELDIR := /lib/modules/$(shell uname -r)/build`: `KERNELDIR` is the location of the prebuilt kernel source. As we said earlier, we need a prebuilt kernel in order to build any module. If you have built your kernel from the source, one should set this variable with the absolute path of the built source directory. `-c` instructs to make utility to change into the specified directory prior to reading the makefiles or doing anything else.
- `M=$(shell pwd)`: This is relevant to the kernel build system. The kernel Makefile uses this variable to locate the directory of the external module to build. Your `.c` files should be placed.
- `all default: modules`: This line instructs the `make` utility to execute the `modules` target, whether `all` or `default` targets, which are classical targets when it comes to building user apps. In other words, `make default` or `make all` or simply `make` commands will be translated into `make modules`.
- `modules modules_install help clean:`: This line represents the list target valid in this Makefile.
- `$(MAKE) -C $(KERNELDIR) M=$(shell pwd) $@`: This is the rule to be executed for each target enumerated above. `$@` will be replaced with the name of the target that caused the rule to run. In other words, if one calls `make modules`, `$@` will be replaced with `modules`, and the rule will become: `$(MAKE) -C`

```
$(KERNELDIR ) M=$(shell pwd) module.
```

In the kernel tree

Before you can build your driver in the kernel tree, you should first identify which directory in drivers should host your `.c` file. Given your file name `mychardev.c`, which contains the source code of your special character driver, it should be placed to the `drivers/char` directory in the kernel source. Every subdirectory in drivers has both `Makefile` and `Kconfig` files.

Add the following content to the `Kconfig` of that directory:

```
config PACKT_MYCDEV
    tristate "Our packtpub special Character driver"
    default m
    help
        Say Y here if you want to support the /dev/mycdev device.
        The /dev/mycdev device is used to access packtpub.
```

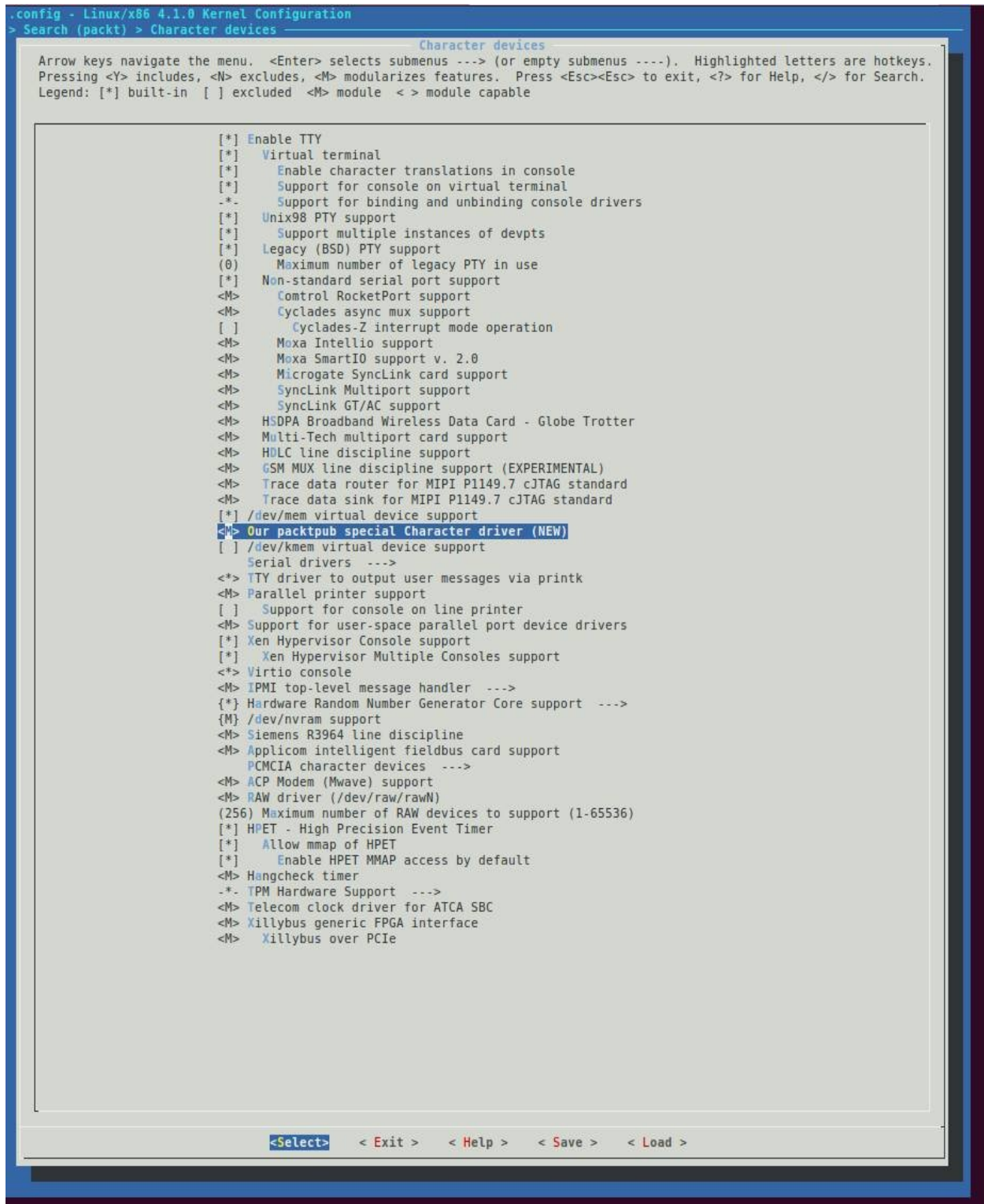
In the makefile of that same directory, add:

```
| obj-$(CONFIG_PACKT_MYCDEV) += mychardev.o
```

Be careful when updating the `Makefile`; the `.o` file name must match the exact name of your `.c` file. If your source file is `foobar.c`, you must use `foobar.o` in the `Makefile`. In order to have your driver built as a module, add the following line in your board `defconfig` in the `arch/arm/configs` directory:
`CONFIG_PACKT_MYCDEV=m`

You may also run `make menuconfig` to select it from the UI, and run `make`, to build the kernel, then `make modules` to build modules (including yours). To make the driver be built in, just replace `m` with `y`: `CONFIG_PACKT_MYCDEV=y`

Everything described here is what embedded board manufacturers do in order to provide a **Board Support Package (BSP)** with their board, with a kernel that already contains their custom drivers:



packt_dev module in kernel tree

Once configured, you can build the kernel with `make` and build modules with `make`

modules.

Modules included in the kernel source tree are installed in

`/lib/modules/${KERNELRELEASE}/kernel/`. On your Linux system, it is `/lib/modules/${uname -r}/kernel/`. Run the following command in order to install the modules: **make modules_install**

sudo apt-get update

sudo apt-get install linux-headers-\$(uname -r)

This will install only headers, not the whole source tree. Headers will then be installed in `/usr/src/linux-headers-$(uname -r)`. On my computer, it is `/usr/src/linux-headers-4.4.0-79-generic/`. There will be a symlink, `/lib/modules/$(uname -r)/build`, pointing to the previously installed headers. It is the path you should specify as your kernel directory in your `Makefile`. It is all you have to do for a prebuilt kernel.

**make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- **

Summary

This chapter showed you the basics of driver development and explained the concept of module/built-in devices, as well as their loading and unloading. Even if you are not able to interact with the user space, you are ready to write a complete driver, print a formatted message, and understand the concept of `init/exit`. The next chapter will deal with character devices, with which you will be able to target enhanced features, write code accessible from the user space, and have a significant impact on the system.

Kernel Facilities and Helper Functions

The kernel is a standalone piece of software, as you'll see in this chapter, that does not make use of any C library. It implements any mechanism you may encounter in modern libraries, and even more, such as compression, string functions, and so on. We will walk step by step through the most important aspects of such capabilities.

In this chapter, we will cover the following topic:

- Introducing the kernel container data structure
- Dealing with the kernel sleeping mechanism
- Using timers
- Delving into the kernel locking mechanism (mutex, spinlock)
- Deferring work using a kernel dedicated API
- Using IRQs

Understanding container_of macro

When it comes to managing several data structures in the code, you'll almost always need to embed one structure into another and retrieve them at any moment without being asked questions about memory offset or boundaries. Let's say you have a `struct person`, as defined here:

```
| struct person {  
|     int age;  
|     char *name;  
| } p;
```

By only having a pointer on `age` or `name`, one can retrieve the whole structure wrapping (containing) that pointer. As the name says, `container_of` macro is used to find the container of the given field of a structure. The macro is defined in `include/linux/kernel.h` and looks like:

```
| #define container_of(ptr, type, member) ({  
|     const typeof(((type *)0)->member) * __mptr = (ptr);  
|     (type *)((char *)__mptr - offsetof(type, member)); })
```

Don't be afraid by the pointers; just see it as:

```
| container_of(pointer, container_type, container_field);
```

Here are the elements of the preceding code fragment:

- `pointer`: This is the pointer to the field in the structure
- `container_type`: This is the type of structure wrapping (containing) the pointer
- `container_field`: This is the name of the field to which `pointer` points inside the structure

Let us consider the following container:

```
| struct person {  
|     int age;  
|     char *name;  
| };
```

Now let us consider one of its instance, along with a pointer to the `name` member:

```
| struct person somebody;
```

```
[...]
char *the_name_ptr = somebody.name;
```

Along with a pointer to the `name` member (`the_name_ptr`), you can use the `container_of` macro in order to get a pointer to the whole structure (container) that wraps this member by using the following:

```
struct person *the_person;
the_person = container_of(the_name_ptr, struct person, name);
```

`container_of` takes the offset of `name` at the beginning of the struct into account to get the correct pointer location. If you subtract the offset of the field `name` from the pointer `the_name_ptr`, you will get the correct location. It is what the macro's last line does:

```
(type *) ( (char *)__mptr - offsetof(type, member) );
```

Applying this to a real example, it gives the following:

```
struct family {
    struct person *father;
    struct person *mother;
    int number_of_suns;
    int salary;
} f;

/*
 * pointer to a field of the structure
 * (could be any member of any family)
 */
struct *person = family.father;
struct family *fam_ptr;

/* now let us retrieve back its family */
fam_ptr = container_of(person, struct family, father);
```

It's all you need to know about the `container_of` macro, and believe me, it is enough. In real drivers that we'll develop further in the book, it looks like the following:

```
struct mcp23016 {
    struct i2c_client *client;
    struct gpio_chip chip;
}

/* retrieve the mcp23016 struct given a pointer 'chip' field */
static inline struct mcp23016 *to_mcp23016(struct gpio_chip *gc)
{
    return container_of(gc, struct mcp23016, chip);
}

static int mcp23016_probe(struct i2c_client *client,
    const struct i2c_device_id *id)
```

```

{
    struct mcp23016 *mcp;
    [...]
    mcp = devm_kzalloc(&client->dev, sizeof(*mcp), GFP_KERNEL);
    if (!mcp)
        return -ENOMEM;
    [...]
}

```

`controller_of` macro is mainly used in generic containers in the kernel. In some examples in this book (starting from [Chapter 5](#), *Platform Device Drivers*), you will encounter the `container_of` macro.

Linked lists

Imagine you have a driver that manages more than one device, let's say five devices. You may need to keep a track of each of them in your driver. What you need here is a linked list. Two types of linked list actually exist:

- Simply linked list
- Doubly linked list

Therefore, kernel developers only implement circular doubly linked lists because this structure allows you to implement FIFO and LIFO, and kernel developers take care to maintain a minimal set of code. The header to be added in the code in order to support lists is `<linux/list.h>`. The data structure at the core of list implementation in the kernel is `struct list_head` structure, defined as the following:

```
struct list_head {  
    struct list_head *next, *prev;  
};
```

The `struct list_head` is used in both the head of the list and each node. In the world of the kernel, before a data structure can be represented as a linked list, that structure must embed a `struct list_head` field. For example, let's create a list of cars:

```
struct car {  
    int door_number;  
    char *color;  
    char *model;  
};
```

Before we can create a list for the car, we must change its structure in order to embed a `struct list_head` field. The structure becomes:

```
struct car {  
    int door_number;  
    char *color;  
    char *model;  
    struct list_head list; /* kernel's list structure */  
};
```

First, we need to create a `struct list_head` variable that will always point to the

head (first element) of our list. This instance of `list_head` is not associated to any car and is special:

```
| static LIST_HEAD(carlist) ;
```

Now we can create cars and add them to our list—`carlist`:

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

struct car *redcar = kmalloc(sizeof(*car), GFP_KERNEL);
struct car *bluecar = kmalloc(sizeof(*car), GFP_KERNEL);

/* Initialize each node's list entry */
INIT_LIST_HEAD(&bluecar->list);
INIT_LIST_HEAD(&redcar->list);

/* allocate memory for color and model field and fill every field */
[...]
list_add(&redcar->list, &carlist) ;
list_add(&bluecar->list, &carlist) ;
```

It is as simple as that. Now, `carlist` contains two elements. Let us get deeper into the linked list API.

Creating and initializing the list

There are two ways to create and initialize the list:

Dynamic method

The dynamic method consists of a `struct list_head` and initializes it with the `INIT_LIST_HEAD` macro:

```
| struct list_head mylist;  
| INIT_LIST_HEAD(&mylist);
```

The following is the expansion of `INIT_LIST_HEAD`:

```
| static inline void INIT_LIST_HEAD(struct list_head *list)  
| {  
|     list->next = list;  
|     list->prev = list;  
| }
```


Static method

Static allocation is done through the `LIST_HEAD` macro: `LIST_HEAD(mylist)`

`LIST_HEADS` definition is defined as follows:

```
| #define LIST_HEAD(name) \  
|     struct list_head name = LIST_HEAD_INIT(name)
```

The following is its expansion:

```
| #define LIST_HEAD_INIT(name) { &(amp;name), &(name) }
```

This assigns each pointer (`prev` and `next`) inside the `name` field to point to `name` itself (just like `INIT_LIST_HEAD` does).

```
struct car *blackcar = kzalloc(sizeof(struct car), GFP_KERNEL);
```

```
/* non static initialization, since it is the embedded list field*/
```

```
INIT_LIST_HEAD(&blackcar->list);
```

As said earlier, use `INIT_LIST_HEAD`, which is a dynamically allocated list and usually part of another structure.

Adding a list node

The kernel provides `list_add` to add a new entry to the list, which is a wrapper around the internal function `__list_add`:

```
void list_add(struct list_head *new, struct list_head *head);
static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

`__list_add` will take two known entries as a parameter, and inserts your elements between them. Its implementation in the kernel is quite easy:

```
static inline void __list_add(struct list_head *new,
                             struct list_head *prev,
                             struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
```

The following is an example of adding two cars in our list:

```
list_add(&redcar->list, &carlist);
list_add(&blue->list, &carlist);
```

This mode can be used to implement a stack. The other function to add an entry into the list is:

```
void list_add_tail(struct list_head *new, struct list_head *head);
```

This inserts the given new entry at the end of the list. Given our previous example, we can use the following:

```
list_add_tail(&redcar->list, &carlist);
list_add_tail(&blue->list, &carlist);
```

This mode can be used to implement a queue.

Deleting a node from the list

List handling is an easy task in kernel code. Deleting a node is straightforward:

```
| void list_del(struct list_head *entry);
```

Following the preceding example, let us delete the red car:

```
| list_del(&redcar->list);
```



list_del disconnects the prev and next pointers of the given entry, resulting in an entry removal. The memory allocated for the node is not freed yet; you need to do that manually with kfree.

```

struct car *acar; /* loop counter */

int blue_car_num = 0;

/* 'list' is the name of the list_head struct in our data structure */
list_for_each_entry(acar, carlist, list){
    if(acar->color == "blue") blue_car_num++; }

#define list_for_each_entry(pos, head, member) \
for (pos = list_entry((head)->next, typeof(*pos), member); \
    &pos->member != (head); \
    pos = list_entry(pos->member.next, typeof(*pos), member))

#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)

```

Given this, we can understand that it is all about container_of's power. Also bear in mind list_for_each_entry_safe(pos, n, head, member).

Kernel sleeping mechanism

Sleeping is the mechanism by which a process relaxes a processor, with the possibility of handling another process. The reason why a processor can sleep could be for sensing data availability, or waiting for a resource to be free.

The kernel scheduler manages a list of tasks to run, known as a run queue. Sleeping processes are not scheduled anymore, since they are removed from that run queue. Unless its state changes (that is, it wakes up), a sleeping process will never be executed. You may relax a processor as soon as one is waiting for something (resource or anything else), and make sure a condition or someone else will wake it up. That said, the Linux kernel eases the implementation of the sleeping mechanism by providing a set of functions and data structures.

```

struct __wait_queue {
    unsigned int flags; #define WQ_FLAG_EXCLUSIVE 0x01
    void *private; wait_queue_func_t func; struct list_head task_list; };

#include <linux/module.h>

#include <linux/init.h>

#include <linux/sched.h>

#include <linux/time.h>

#include <linux/delay.h>

#include<linux/workqueue.h>


static DECLARE_WAIT_QUEUE_HEAD(my_wq); static int
condition = 0;


/* declare a work queue*/

static struct work_struct wrk;


static void work_handler(struct work_struct *work) {

    printk("Waitqueue module handler %s\n", __FUNCTION__);
    msleep(5000); printk("Wake up the sleeping module\n"); condition =

```

```
1; wake_up_interruptible(&my_wq); }
```

```
static int __init my_init(void)
```

```
{
```

```
    printk("Wait queue example\n");
```

```
    INIT_WORK(&wrk, work_handler); schedule_work(&wrk);
```

```
    printk("Going to sleep %s\n", __FUNCTION__);  
    wait_event_interruptible(my_wq, condition != 0);
```

```
    pr_info("woken up by the work job\n"); return 0;
```

```
}
```

```
void my_exit(void)
```

```
{
```

```
    printk("waitqueue example cleanup\n"); }
```

```
module_init(my_init);
```

```
module_exit(my_exit);
```

```
MODULE_AUTHOR("John Madieu <john.madieu@foobar.com>");  
MODULE_LICENSE("GPL");
```


[342081.385491] Wait queue example
[342081.385505] Going to sleep my_init
[342081.385515] Waitqueue module handler work_handler
[342086.387017] Wake up the sleeping module
[342086.387096] woken up by the work job
[342092.912033] waitqueue example cleanup

Delay and timer management

Time is one of the most used resources, right after memory. It is used to do almost everything: defer work, sleep, scheduling, timeout, and many other tasks.

There are the two categories of time. The kernel uses absolute time to know what time it is, that is, the date and time of the day, whereas relative time is used by, for example, the kernel scheduler. For absolute time, there is a hardware chip called **real-time clock (RTC)**. We will deal with such devices later in the book in [Chapter 18](#), *RTC Drivers*. On the other side, to handle relative time, the kernel relies on a CPU feature (peripheral), called a timer, which, from the kernel's point of view, is called a *kernel timer*. Kernel timers are what we will talk about in this section.

Kernel timers are classified into two different parts:

- Standard timers, or system timers
- High-resolution timers

Standard timers

Standard timers are kernel timers operating on the granularity of jiffies.

Jiffies and HZ

A jiffy is a kernel unit of time declared in `<linux/jiffies.h>`. To understand jiffies, we need to introduce a new constant HZ, which is the number of times `jiffies` is incremented in one second. Each increment is called a *tick*. In other words, HZ represents the size of a jiffy. HZ depends on the hardware and on the kernel version, and also determines how frequently the clock interrupt fires. This is configurable on some architecture, fixed on other ones.

What it means is that `jiffies` is incremented HZ times every second. If HZ = 1,000, then it is incremented 1,000 times (that is, one tick every 1/1,000 seconds). Once defined, the **programmable interrupt timer (PIT)**, which is a hardware component, is programmed with that value in order to increment jiffies when the PIT interrupt comes in.

Depending on the platform, jiffies can lead to overflow. On a 32-bit system, HZ = 1,000 will result in about 50 days duration only, whereas the duration is about 600 million years on a 64-bit system. By storing jiffies in a 64-bit variable, the problem is solved. A second variable has then been introduced and defined in `<linux/jiffies.h>`: `extern u64 jiffies_64;`

In this manner on 32-bit systems, `jiffies` will point to low-order 32-bits, and `jiffies_64` will point to high-order bits. On 64-bit platforms, `jiffies = jiffies_64`.

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

```
struct timer_list {
```

```
    struct list_head entry; unsigned long expires; struct tvec_t_base_s  
    *base; void (*function)(unsigned long); unsigned long data; );
```

expires is an absolute value in jiffies. entry is a doubly linked list,
and data is optional, and passed to the callback function.

Timer setup initialization

The following are steps to initialize timers:

1. **Setting up the timer:** Set up the timer, feeding the user-defined callback and data:

```
void setup_timer( struct timer_list *timer, \
                  void (*function)(unsigned long), \
                  unsigned long data);
```

One can also use this:

```
void init_timer(struct timer_list *timer);
```

`setup_timer` is a wrapper around `init_timer`.

2. **Setting the expiration time:** When the timer is initialized, we need to set its expiration before the callback gets fired:

```
int mod_timer( struct timer_list *timer, unsigned long expires);
```

3. **Releasing the timer:** When you are done with the timer, it needs to be released:

```
void del_timer(struct timer_list *timer);
int del_timer_sync(struct timer_list *timer);
```

`del_timer` returns void whether it has deactivated a pending timer or not. Its return value is 0 on an inactive timer, or 1 on an active one. The last, `del_timer_sync`, waits for the handler to finish its execution, even those that may happen on another CPU. You should not hold a lock preventing the handler's completion, otherwise it will result in a dead lock. You should release the timer in the module cleanup routine. You can independently check whether the timer is running or not:

```
int timer_pending( const struct timer_list *timer);
```

This function checks whether there are any fired timer callbacks pending.

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

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

```
static struct timer_list my_timer;
```

```
void my_timer_callback(unsigned long data) {
```

```
    printk("%s called (%ld).\n", __FUNCTION__, jiffies); }
```

```
static int __init my_init(void) {
```

```
    int retval; printk("Timer module loaded\n");
```

```
    setup_timer(&my_timer, my_timer_callback, 0); printk("Setup  
timer to fire in 300ms (%ld)\n", jiffies);
```

```
    retval = mod_timer( &my_timer, jiffies + msecs_to_jiffies(300) ); if  
(retval) printk("Timer firing failed\n");
```

```
    return 0; }
```

```
static void my_exit(void)
```

```
{
```

```
    int retval; retval = del_timer(&my_timer); /* Is timer still active (1)  
or no (0) */
```

```
if (retval) printk("The timer is still in use...\n");
```

```
pr_info("Timer module unloaded\n"); }
```

```
module_init(my_init);
```

```
module_exit(my_exit);
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
```

```
MODULE_DESCRIPTION("Standard timer example");
```

```
MODULE_LICENSE("GPL");
```


High resolution timers (HRTs)

Standard timers are less accurate and do not suit real-time applications. High-resolution timers, introduced in kernel v2.6.16 (and enabled by the `CONFIG_HIGH_RES_TIMERS` option in the kernel configuration) have a resolution of microseconds (up to nanoseconds, depending on the platform), compared to milliseconds on standard timers. The standard timer depends on HZ (since they rely on jiffies), whereas HRT implementation is based on `ktime`.

Kernel and hardware must support an HRT before being used on your system. In other words, there must be an arch-dependent code implemented to access your hardware HRTs.

HRT API

The required headers are:

```
| #include <linux/hrtimer.h>
```

An HRT is represented in the kernel as an instance of `hrtimer`:

```
| struct hrtimer {  
|     struct timerqueue_node node;  
|     ktime_t _softexpires;  
|     enum hrtimer_restart (*function)(struct hrtimer *);  
|     struct hrtimer_clock_base *base;  
|     u8 state;  
|     u8 is_rel;  
| };
```

HRT setup initialization

1. **Initializing the hrtimer:** Before hrtimer initialization, you need to set up a `ktime`, which represents time duration. We will see how to achieve that in the following example:

```
void hrtimer_init( struct hrtimer *time, clockid_t which_clock,
                  enum hrtimer_mode mode);
```

2. **Starting hrtimer:** hrtimer can be started as shown in the following example:

```
int hrtimer_start( struct hrtimer *timer, ktime_t time,
                  const enum hrtimer_mode mode);
```

`mode` represents the expiry mode. It should be `HRTIMER_MODE_ABS` for an absolute time value, or `HRTIMER_MODE_REL` for a time value relative to now.

3. **hrtimer cancellation:** You can either cancel the timer or see whether it is possible to cancel it or not:

```
int hrtimer_cancel( struct hrtimer *timer);
int hrtimer_try_to_cancel(struct hrtimer *timer);
```

Both return `0` when the timer is not active and `1` when the timer is active. The difference between these two functions is that `hrtimer_try_to_cancel` fails if the timer is active or its callback is running, returning `-1`, whereas `hrtimer_cancel` will wait until the callback finishes.

We can independently check whether the hrtimer's callback is still running with the following:

```
int hrtimer_callback_running(struct hrtimer *timer);
```

Remember, `hrtimer_try_to_cancel` internally calls `hrtimer_callback_running`.



In order to prevent the timer from automatically restarting, the hrtimer callback function must return `HRTIMER_NORESTART`.

You can check whether HRTs are available on your system by doing the following:

- By looking in the kernel config file, which should contain something like `CONFIG_HIGH_RES_TIMERS=y`: `zcat /proc/configs.gz | grep CONFIG_HIGH_RES_TIMERS`.
- By looking at the `cat /proc/timer_list` OR `cat /proc/timer_list | grep resolution` result. The `.resolution` entry must show 1 nsecs and the `event_handler` must show `hrtimer_interrupts`.
- By using the `clock_getres` system call.
- From within the kernel code, by using `#ifdef CONFIG_HIGH_RES_TIMERS`.

With HRTs enabled on your system, the accuracy of sleep and timer system calls do not depend on jiffies anymore, but they are still as accurate as HRTs are. It is the reason why some systems do not support `nanosleep()`, for example.

Dynamic tick/tickless kernel

With previous HZ options, the kernel is interrupted HZ times per second in order to reschedule tasks, even in an idle state. If HZ is set to 1,000, there will be 1,000 kernel interruptions per second, preventing the CPU from being idle for a long time, thus affecting CPU power consumption.

Now let's look at a kernel with no fixed or predefined ticks, where the ticks are disabled until some task needs to be performed. We call such a kernel a **tickless kernel**. In fact, tick activation is scheduled, based on the next action. The right name should be **dynamic tick kernel**. The kernel is responsible for task scheduling, and maintains a list of runnable tasks (the run queue) in the system. When there is no task to schedule, the scheduler switches to the idle thread, which enables dynamic tick by disabling the periodic tick until the next timer expires (a new task is queued for processing).

Under the hood, the kernel also maintains a list of the tasks timeouts (it then knows when and how long it has to sleep). In an idle state, if the next tick is further away than the lowest timeout in the tasks list timeout, the kernel programs the timer with that timeout value. When the timer expires, the kernel re-enables the periodic ticks back and invokes the scheduler, which then schedules the task associated with the timeout. This is how the tickless kernel removes the periodic tick and saves power when idle.

Delays and sleep in the kernel

Without going deep into the details, there are two types of delays, depending on the context your code runs in: atomic or nonatomic. The mandatory header to handle delays in the kernel is `#include <linux/delay>`.

Atomic context

Tasks in the atomic context (such as ISR) can't sleep, and can't be scheduled; it is the reason why busy-wait loops are used for delaying purposes in an atomic context. The kernel exposes the `xdelay` family of functions that will spend time in a busy loop, long (based on jiffies) enough to achieve the desired delay:

- `ndelay(unsigned long nsecs)`
- `udelay(unsigned long usecs)`
- `mdelay(unsigned long msecs)`

You should always use `udelay()` since `ndelay()` precision depends on how accurate your hardware timer is (not always the case on an embedded SOC). Use of `mdelay()` is also discouraged.

Timer handlers (callbacks) are executed in an atomic context, meaning that sleeping is not allowed at all. By *sleeping*, I mean any function that may result in sending the caller to sleep, such as allocating memory, locking a mutex, an explicit call to `sleep()` function, and so on.

Nonatomic context

In a nonatomic context, the kernel provides the `sleep[_range]` family of functions and which function to use depends on how long you need to delay by:

- `udelay(unsigned long usecs)`: Busy-wait loop based. You should use this function if you need to sleep for a few μ secs ($< \sim 10$ us).
- `usleep_range(unsigned long min, unsigned long max)`: Relies on hrtimers, and it is recommended to let this sleep for few $\sim \mu$ secs or small msecs (10 us - 20 ms), avoiding the busy-wait loop of `udelay()`.
- `msleep(unsigned long msecs)`: Backed by jiffies/legacy_timers. You should use this for larger, msecs sleep (10 ms+).



*Sleep and delay topics are well explained in
Documentation/timers/timers-howto.txt in the kernel source.*

Kernel locking mechanism

Locking is a mechanism that helps share resources between different threads or processes. A shared resource is a data or a device that can be accessed by at least two users, simultaneously or no. Locking mechanisms prevent abusive access, for example, a process writing data when another one is reading in the same place, or two processes accessing the same device (the same GPIO for example). The kernel provides several locking mechanisms. The most important are:

- Mutex
- Semaphore
- Spinlock

We will only learn about mutexes and spinlock, since they are widely used in device drivers.

Mutex

Mutual exclusion (mutex) is the de facto most used locking mechanism. To understand how it works, let's see what its structure looks like in

```
include/linux/mutex.h: struct mutex { /* 1: unlocked, 0: locked, negative: locked,  
possible waiters */ atomic_t count; spinlock_t wait_lock; struct list_head  
wait_list; [...] };
```

As we have seen in the section *wait queue*, there is also a `list` type field in the structure: `wait_list`. The principle of sleeping is the same.

Contenders are removed from the scheduler run queue and put onto the wait list (`wait_list`) in a sleep state. The kernel then schedules and executes other tasks. When the lock is released, a waiter in the wait queue is woken, moved off the `wait_list`, and scheduled back.

Mutex API

Using mutex requires only a few basic functions:

Declare

- Statically:

```
| DEFINE_MUTEX(my_mutex);
```

- Dynamically:

```
| struct mutex my_mutex;  
| mutex_init(&my_mutex);
```

Acquire and release

- Lock:

```
void mutex_lock(struct mutex *lock);  
int  mutex_lock_interruptible(struct mutex *lock);  
int  mutex_lock_killable(struct mutex *lock);
```

- Unlock:

```
void mutex_unlock(struct mutex *lock);
```

Sometimes, you may only need to check whether a mutex is locked or not. For that purpose, you can use the `int mutex_is_locked(struct mutex *lock)` function.

```
int mutex_is_locked(struct mutex *lock);
```

What this function does is just check whether the mutex's owner is empty (`NULL`) or not. There is also `mutex_trylock`, that acquires the mutex if it is not already locked, and returns `1`; otherwise, it returns `0`:

```
int mutex_trylock(struct mutex *lock);
```

As with the wait queue's interruptible family function, `mutex_lock_interruptible()`, which is recommended, will result in the driver being able to be interrupted by any signal, whereas with `mutex_lock_killable()`, only signals killing the process can interrupt the driver.

You should be very careful with `mutex_lock()`, and use it when you can guarantee that the mutex will be released, whatever happens. In the user context, it is recommended you always use `mutex_lock_interruptible()` to acquire the mutex, since `mutex_lock()` will not return if a signal is received (even a `ctrl + c`).

Here is an example of a mutex implementation:

```
struct mutex my_mutex;  
mutex_init(&my_mutex);  
  
/* inside a work or a thread */  
mutex_lock(&my_mutex);  
access_shared_memory();  
mutex_unlock(&my_mutex);
```

Please have a look at `include/linux/mutex.h` in the kernel source to see the strict rules you must respect with mutexes. Here are some of them:

- Only one task can hold the mutex at a time; this is actually not a rule, but a fact
- Multiple unlocks are not permitted
- They must be initialized through the API
- A task holding the mutex may not exit, since the mutex will remain locked, and possible contenders will wait (will sleep) forever
- Memory areas where held locks reside must not be freed
- Held mutexes must not be reinitialized
- Since they involve rescheduling, mutexes may not be used in atomic contexts, such as tasklets and timers



As with `wait_queue`, there is no polling mechanism with mutexes. Every time that `mutex_unlock` is called on a mutex, the kernel checks for waiters in `wait_list`. If any, one (and only one) of them is awakened and scheduled; they are woken in the same order in which they were put to sleep.

Spinlock

Like mutex, spinlock is a mutual exclusion mechanism; it only has two states:

- locked (acquired)
- unlocked (released)

Any thread that needs to acquire the spinlock will active loop until the lock is acquired, which breaks out of the loop. This is the point where mutex and spinlock differ. Since spinlock heavily consumes the CPU while looping, it should be used for very quick acquires, especially when time to hold the spinlock is less than time to reschedule. Spinlock should be released as soon as the critical task is done.

In order to avoid wasting CPU time by scheduling a thread that may probably spin, trying to acquire a lock held by another thread moved off the run queue, the kernel disables preemption whenever a code holding a spinlock is running. With preemption disabled, we prevent the spinlock holder from being moved off the run queue, which could lead waiting processes to spin for a long time and consume CPU.

As long as one holds a spinlock, other tasks may be spinning while waiting on it. By using spinlock, you asserts and guarantee that it will not be held for a long time. You can say it is better to spin in a loop, wasting CPU time, than the cost of sleeping your thread, context-shifting to another thread or process, and being woken up afterward. Spinning on a processor means no other task can run on that processor; it then makes no sense to use spinlock on a single core machine. In the best case, you will slow down the system; in the worst case, you will deadlock, as with mutexes. For this reason, the kernel just disables preemption in response to the `spin_lock(spinlock_t *lock)` function on single processor. On a single processor (core) system, you should use `spin_lock_irqsave()` and `spin_unlock_irqrestore()`, which will respectively disable the interrupts on the CPU, preventing interrupt concurrency.

Since you do not know in advance what system you will write the driver for, it is recommended you acquire a spinlock using `spin_lock_irqsave(spinlock_t *lock,`

unsigned long flags), which disables interrupts on the current processor (the processor where it is called) before taking the spinlock. `spin_lock_irqsave` internally calls `local_irq_save(flags);`, an architecture-dependent function to save the IRQ status, and `preempt_disable()` to disable preemption on the relevant CPU. You should then release the lock with `spin_unlock_irqrestore()`, which does the reverse operations that we previously enumerated. This is a code that does lock acquire and release. It is an IRQ handler, but let's just focus on the lock aspect. We will discuss more about IRQ handlers in the next section:

```
/* some where */
spinlock_t my_spinlock;
spin_lock_init(&my_spinlock);

static irqreturn_t my_irq_handler(int irq, void *data)
{
    unsigned long status, flags;

    spin_lock_irqsave(&my_spinlock, flags);
    status = access_shared_resources();

    spin_unlock_irqrestore(&my_spinlock, flags);
    return IRQ_HANDLED;
}
```


Spinlock versus mutexes

Used for concurrency in the kernel, spinlocks and mutexes each have their own objectives:

- Mutexes protect the process's critical resource, whereas spinlock protects the IRQ handler's critical sections
- Mutexes put contenders to sleep until the lock is acquired, whereas spinlocks infinitely spin in a loop (consuming CPU) until the lock is acquired
- Because of the previous point, you can't hold spinlock for a long time, since waiters will waste CPU time waiting for the lock, whereas a mutex can be held as long as the resource needs to be protected, since contenders are put to sleep in a wait queue



When dealing with spinlocks, please keep in mind that preemption is disabled only for threads holding spinlocks, not for spinning waiters.

Work deferring mechanism

Deferring is a method by which you schedule a piece of work to be executed in the future. It's a way to report an action later. Obviously, the kernel provides facilities to implement such a mechanism; it allows you to defer functions, whatever their type, to be called and executed later. There are three of them in the kernel:

- **SoftIRQs:** Executed in an atomic context
- **Tasklets:** Executed in an atomic context
- **Workqueues:** Executed in a process context

Softirqs and ksoftirqd

Software IRQ (softirq), or software interrupt is a deferring mechanism used only for very fast processing, since it runs with a disabled scheduler (in an interrupt context). You'll rarely (almost never) want to deal with softirq directly. There are only networks and block device subsystems using softirq. Tasklets are an instantiation of softirqs, and will be sufficient in almost every case that you feel the need to use softirqs.

ksoftirqd

In most cases, softirqs are scheduled in hardware interrupts, which may arrive very quickly, faster than they can be serviced. They are then queued by the kernel in order to be processed later. **Ksoftirqds** are responsible for late execution (process context this time). A ksoftirqd is a per-CPU kernel thread raised to handle unserviced software interrupts:

```
jma@jma:~$ top
top - 16:24:48 up 22:21, 1 user, load average: 0.60, 0.65, 0.50
Tasks: 315 total, 1 running, 314 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.2 us, 5.9 sy, 0.0 ni, 92.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 16328448 total, 4970168 free, 1947672 used, 9410608 buff/cache
KiB Swap: 16672764 total, 16622644 free, 50120 used, 7623576 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR   %CPU  %MEM     TIME+ COMMAND
31795 jma        20   0 8058284  6.381g  6.188g   42.9  41.0   45:11.31 vmware-vmx
1286 root        20   0 424440  181272  61612   4.3   1.1   32:36.91 Xorg
2304 jma        20   0 1757216  335860  83464   2.7   2.1   19:28.33 compiz
29216 jma        20   0 755508  240316  89356   2.3   1.5   3:02.26 skype
32554 jma        20   0 657824  38472  28796   1.0   0.2   0:01.79 gnome-terminal-
1224 root      -51   0      0      0      0   0.7   0.0   6:17.17 irq/139-nvidia
531 root      -51   0      0      0      0   0.3   0.0   0:11.80 irq/141-iwlwifi
2987 jma        20   0 42092  4104  3200   0.3   0.0   0:00.03 top
30410 jma        20   0 1014808 105680  67860   0.3   0.6   0:09.12 vmlayer
31893 jma        20   0 93624  8380  6568   0.3   0.1   0:00.39 thnucnt
1 root      20   0 120084  5556  3700   0.0   0.0   0:02.10 systemd
2 root      20   0      0      0      0   0.0   0.0   0:00.03 kthreadd
3 root      20   0      0      0      0   0.0   0.0   0:00.06 ksoftirqd/0
5 root      0 -20      0      0      0   0.0   0.0   0:00.00 kworker/0:0H
7 root      20   0      0      0      0   0.0   0.0   1:51.71 rcu_sched
8 root      20   0      0      0      0   0.0   0.0   0:00.00 rcu_bh
9 root      rt   0      0      0      0   0.0   0.0   0:00.05 migration/0
10 root     rt   0      0      0      0   0.0   0.0   0:00.30 watchdog/0
11 root     rt   0      0      0      0   0.0   0.0   0:00.28 watchdog/1
12 root     rt   0      0      0      0   0.0   0.0   0:00.04 migration/1
13 root     20   0      0      0      0   0.0   0.0   0:00.05 ksoftirqd/1
15 root     0 -20      0      0      0   0.0   0.0   0:00.00 kworker/1:0H
16 root     rt   0      0      0      0   0.0   0.0   0:00.27 watchdog/2
17 root     rt   0      0      0      0   0.0   0.0   0:00.04 migration/2
18 root     20   0      0      0      0   0.0   0.0   0:00.04 ksoftirqd/2
20 root     0 -20      0      0      0   0.0   0.0   0:00.00 kworker/2:0H
21 root     rt   0      0      0      0   0.0   0.0   0:00.29 watchdog/3
22 root     rt   0      0      0      0   0.0   0.0   0:00.03 migration/3
23 root     20   0      0      0      0   0.0   0.0   0:00.06 ksoftirqd/3
25 root     0 -20      0      0      0   0.0   0.0   0:00.00 kworker/3:0H
26 root     rt   0      0      0      0   0.0   0.0   0:00.26 watchdog/4
27 root     rt   0      0      0      0   0.0   0.0   0:00.05 migration/4
28 root     20   0      0      0      0   0.0   0.0   0:00.02 ksoftirqd/4
30 root     0 -20      0      0      0   0.0   0.0   0:00.00 kworker/4:0H
31 root     rt   0      0      0      0   0.0   0.0   0:00.27 watchdog/5
32 root     rt   0      0      0      0   0.0   0.0   0:00.06 migration/5
33 root     20   0      0      0      0   0.0   0.0   0:00.04 ksoftirqd/5
35 root     0 -20      0      0      0   0.0   0.0   0:00.00 kworker/5:0H
36 root     rt   0      0      0      0   0.0   0.0   0:00.24 watchdog/6
37 root     rt   0      0      0      0   0.0   0.0   0:00.05 migration/6
38 root     20   0      0      0      0   0.0   0.0   0:00.02 ksoftirqd/6
40 root     0 -20      0      0      0   0.0   0.0   0:00.00 kworker/6:0H
41 root     rt   0      0      0      0   0.0   0.0   0:00.24 watchdog/7
42 root     rt   0      0      0      0   0.0   0.0   0:00.05 migration/7
43 root     20   0      0      0      0   0.0   0.0   0:00.02 ksoftirqd/7
45 root     0 -20      0      0      0   0.0   0.0   0:00.00 kworker/7:0H
46 root     20   0      0      0      0   0.0   0.0   0:00.00 kdevtmpfs
47 root     0 -20      0      0      0   0.0   0.0   0:00.00 netns
48 root     0 -20      0      0      0   0.0   0.0   0:00.00 perf
49 root     20   0      0      0      0   0.0   0.0   0:00.06 khungtaskd
50 root     0 -20      0      0      0   0.0   0.0   0:00.00 writeback
51 root     25   5      0      0      0   0.0   0.0   0:00.00 ksm
52 root     39  19      0      0      0   0.0   0.0   0:00.85 khugepaged
53 root     0 -20      0      0      0   0.0   0.0   0:00.00 crypto
54 root     0 -20      0      0      0   0.0   0.0   0:00.00 kintegrityd
55 root     0 -20      0      0      0   0.0   0.0   0:00.00 bioset
56 root     0 -20      0      0      0   0.0   0.0   0:00.00 kblockd
57 root     0 -20      0      0      0   0.0   0.0   0:00.00 ata_sff
58 root     0 -20      0      0      0   0.0   0.0   0:00.00 md
59 root     0 -20      0      0      0   0.0   0.0   0:00.00 devfreq_wq
63 root     20   0      0      0      0   0.0   0.0   0:02.30 kswapd0
64 root     0 -20      0      0      0   0.0   0.0   0:00.00 vmstat
65 root     20   0      0      0      0   0.0   0.0   0:00.00 fsnotify_mark
66 root     20   0      0      0      0   0.0   0.0   0:00.00 ecryptfs-kthrea
82 root     0 -20      0      0      0   0.0   0.0   0:00.00 kthrotld
83 root     0 -20      0      0      0   0.0   0.0   0:00.00 acpi_thermal_pm
86 root     0 -20      0      0      0   0.0   0.0   0:00.00 bioset
87 root     0 -20      0      0      0   0.0   0.0   0:00.00 bioset
88 root     0 -20      0      0      0   0.0   0.0   0:00.00 bioset
```

In the preceding `top` sample from my personal computer, you can see `ksoftirqd/n` entries, where `n` is the CPU number that the `ksoftirqd` runs on. CPU-consuming `ksoftirqd` may indicate an overloaded system or a system under **interrupts**

storm, which is never good. You can have a look at `kernel/softirq.c` to see how `ksoftirqds` are designed.

```
struct tasklet_struct {
```

```
    struct tasklet_struct *next; unsigned long state; atomic_t count; void  
    (*func)(unsigned long); unsigned long data; };
```

Tasklets are not re-entrant by nature. A code is called reentrant if it can be interrupted anywhere in the middle of its execution, and then be safely called again. Tasklets are designed such that a tasklet can run on one and only one CPU simultaneously (even on an SMP system), which is the CPU it was scheduled on, but different tasklets may be run simultaneously on different CPUs. The tasklet API is quite basic and intuitive.

```
#define DECLARE_TASKLET(name, func, data) \
```

```
    struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func,  
    data }
```

```
#define DECLARE_TASKLET_DISABLED(name, func, data) \
```

```
    struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func,  
    data }
```

Globally, setting the count field to 0 means that the tasklet is disabled and cannot be executed, whereas a nonzero value means the opposite.

Enabling and disabling a tasklet

There is one function to enable a tasklet: `void tasklet_enable(struct tasklet_struct *)`;

`tasklet_enable` simply enables the tasklet. In older kernel versions, you may find `void tasklet_hi_enable(struct tasklet_struct *)` is used, but those two functions do exactly the same thing. To disable a tasklet, call: `void tasklet_disable(struct tasklet_struct *)`;

You can also call:

```
| void tasklet_disable_nosync(struct tasklet_struct *);
```

`tasklet_disable` will disable the tasklet and return only when the tasklet has terminated its execution (if it was running), whereas `tasklet_disable_nosync` returns immediately, even if the termination has not occurred.


```
void tasklet_schedule(struct tasklet_struct *t); void  
tasklet_hi_schedule(struct tasklet_struct *t);
```

```
void tasklet_kill(struct tasklet_struct *t);
```

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

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

```
#include <linux/interrupt.h> /* for tasklets API */
```

```
char tasklet_data[]="We use a string; but it could be pointer to a  
structure";
```

```
/* Tasklet handler, that just print the data */
```

```
void tasklet_work(unsigned long data) {
```

```
    printk("%s\n", (char *)data); }
```

```
DECLARE_TASKLET(my_tasklet, tasklet_function, (unsigned long)  
tasklet_data);
```

```
static int __init my_init(void)
```

```
{
```

```
    /*
```

```
    * Schedule the handler.
```

```
* Tasklet arealso scheduled from interrupt handler */  
  
tasklet_schedule(&my_tasklet); return 0;  
  
}  
  
void my_exit(void)  
{  
  
    tasklet_kill(&my_tasklet); }  
  
module_init(my_init);  
  
module_exit(my_exit);  
  
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");  
MODULE_LICENSE("GPL");
```

Work queues

Added since Linux kernel 2.6, the most used and simple deferring mechanism is the work queue. It is the last one we will talk about in this chapter. As a deferring mechanism, it takes an opposite approach to the others we've seen, running only in a preemptible context. It is the only choice when you need to sleep in your bottom half (I will explain what a bottom half is later in the next section). By sleep, I mean process I/O data, hold mutexes, delay, and all the other tasks that may lead to sleep or move the task off the run queue.

Keep in mind that work queues are built on top of kernel threads, and this is the reason why I decided not to talk about the kernel thread as a deferring mechanism at all. However, there are two ways to deal with work queues in the kernel. First, there is a default shared work queue, handled by a set of kernel threads, each running on a CPU. Once you have work to schedule, you queue that work into the global work queue, which will be executed at the appropriate moment. The other method is to run the work queue in a dedicated kernel thread. It means whenever your work queue handler needs to be executed, your kernel thread is woken up to handle it, instead of one of the default predefined threads.

Structures and functions to call are different, depending on whether you chose a shared work queue or dedicated ones.

```
struct workqueue_struct *system_wq __read_mostly;
EXPORT_SYMBOL(system_wq);

void flush_scheduled_work(void);

#include <linux/module.h>

#include <linux/init.h>

#include <linux/sched.h> /* for sleep */

#include <linux/wait.h> /* for wait queue */

#include <linux/time.h>

#include <linux/delay.h>

#include <linux/slab.h> /* for kmalloc() */

#include <linux/workqueue.h>

//static DECLARE_WAIT_QUEUE_HEAD(my_wq); static int sleep
= 0;

struct work_data {

    struct work_struct my_work; wait_queue_head_t my_wq; int
the_data; };

static void work_handler(struct work_struct *work) {
```

```

    struct work_data *my_data = container_of(work, \
        struct work_data, my_work); printk("Work queue module handler:
%s, data is %d\n", __FUNCTION__, my_data->the_data);
msleep(2000); wake_up_interruptible(&my_data->my_wq);
kfree(my_data); }

```

```

static int __init my_init(void)

```

```

{

```

```

    struct work_data * my_data;

```

```

    my_data = kmalloc(sizeof(struct work_data), GFP_KERNEL);
my_data->the_data = 34;

```

```

    INIT_WORK(&my_data->my_work, work_handler);
init_waitqueue_head(&my_data->my_wq);

```

```

    schedule_work(&my_data->my_work); printk("I'm goint to sleep
...\n"); wait_event_interruptible(my_data->my_wq, sleep != 0);
printk("I am Waked up...\n"); return 0;

```

```

}

```

```

static void __exit my_exit(void)

```

```

{

```

```

    printk("Work queue module exit: %s %d\n", __FUNCTION__,
__LINE__); }

```

```
module_init(my_init);
```

```
module_exit(my_exit);
```

```
MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com> ");
```

```
MODULE_DESCRIPTION("Shared workqueue");
```

In order to pass data to my work queue handler, you may have noticed that in both examples, I've embedded my `work_struct` structure inside my custom data structure, and used `container_of` to retrieve it. It is the common way to pass data to the work queue handler.

Dedicated work queue

Here, the work queue is represented as an instance of `struct workqueue_struct`. The work to be queued into the work queue is represented as an instance of `struct work_struct`. There are four steps involved prior to scheduling your work in your own kernel thread:

1. Declare/initialize a `struct workqueue_struct`.
2. Create your work function.
3. Create a `struct work_struct` so that your work function will be embedded into it.
4. Embed your work function in the `work_struct`.

```
queue_delayed_work(myqueue, &thework, <delay>);
```

```
if ( !cancel_delayed_work( &thework) )  
{<br/>flush_workqueue(myqueue);  
<br/>destroy_workqueue(myqueue);<br/>}
```

```
INIT_DELAYED_WORK(_work, _func);
```

```
INIT_DELAYED_WORK_DEFERRABLE(_work, _func);
```

```
int queue_delayed_work(struct workqueue_struct *wq, struct  
delayed_work *dwork, unsigned long delay)
```

```
int queue_work_on(int cpu, struct workqueue_struct *wq, struct  
work_struct *work);
```

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

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

```
#include <linux/workqueue.h> /* for work queue */
```

```
#include <linux/slab.h> /* for kmalloc() */
```

```
struct workqueue_struct *wq;
```

```
struct work_data {
```

```
    struct work_struct my_work; int the_data; };
```



```
static void work_handler(struct work_struct *work) {  
  
    struct work_data * my_data = container_of(work, struct work_data,  
    my_work); printk("Work queue module handler: %s, data is %d\n",  
    __FUNCTION__, my_data->the_data); kfree(my_data); }
```

```
static int __init my_init(void)  
  
{  
  
    struct work_data * my_data;  
  
    printk("Work queue module init: %s %d\n", __FUNCTION__,  
    __LINE__); wq =  
    create_singlethread_workqueue("my_single_thread"); my_data =  
    kmalloc(sizeof(struct work_data), GFP_KERNEL);  
  
    my_data->the_data = 34; INIT_WORK(&my_data->my_work,  
    work_handler); queue_work(wq, &my_data->my_work);  
  
    return 0;  
  
}
```

```
static void __exit my_exit(void)  
  
{  
  
    flush_workqueue(wq); destroy_workqueue(wq); printk("Work  
queue module exit: %s %d\n", __FUNCTION__, __LINE__); }
```

```
module_init(my_init);
```

```
module_exit(my_exit);
```

```
MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
```

Predefined (shared) workqueue and standard workqueue functions

The predefined work queue is defined in `kernel/workqueue.c` as follows: `struct workqueue_struct *system_wq __read_mostly;`

It is nothing more than a standard work for which the kernel provides a custom API that simply wraps around the standard one.

Comparisons between kernel predefined work queue functions and standard work queue functions are mentioned as follows:

Predefined work queue function	Equivalent standard work queue function
<code>schedule_work(w)</code>	<code>queue_work(keventd_wq,w)</code>
<code>schedule_delayed_work(w,d)</code>	<code>queue_delayed_work(keventd_wq,w,d)</code> (on any CPU)
<code>schedule_delayed_work_on(cpu,w,d)</code>	<code>queue_delayed_work(keventd_wq,w,d)</code> (on a given CPU)
<code>flush_scheduled_work()</code>	<code>flush_workqueue(keventd_wq)</code>

Kernel threads

Work queues run on top of kernel threads. You already use kernel threads when you use work queues. It is the reason why I have decided not to talk about the kernel thread API.

Kernel interruption mechanism

An interrupt is the way a device halts the kernel, telling it that something interesting or important has happened. These are called IRQs on Linux systems. The main advantage interrupts offer is to avoid devices polling. It is up to the device to tell if there is a change in its state; it is not up to us to poll it.

In order to get notified when an interrupt occurs, you need to register to that IRQ, providing a function called interrupt handler that will be called every time that interrupt is raised.

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long
flags, const char *name, void *dev)
```

```
struct my_data {
```

```
    struct input_dev *idev; struct i2c_client *client; char name[64];
char phys[32]; };
```

```
static irqreturn_t my_irq_handler(int irq, void *dev_id) {
```

```
    struct my_data *md = dev_id; unsigned char nextstate =
read_state(lp); /* Check whether my device raised the irq or no */
```

```
    [...]
```

```
    return IRQ_HANDLED; }
```

```
/* some where in the code, in the probe function */
```

```
int ret;
```

```
struct my_data *md;
```

```
md = kzalloc(sizeof(*md), GFP_KERNEL);
```

```
ret = request_irq(client->irq, my_irq_handler,
IRQF_TRIGGER_LOW | IRQF_ONESHOT, DRV_NAME, md);
```

```
/* far in the release function */
```

```
free_irq(client->irq, md);
```

```
void free_irq(unsigned int irq, void *dev)
```

If the specified IRQ is not shared, `free_irq` will not only remove the handler, but will also disable the line. If it is shared, only the handler identified through `dev` (which should be the same as that used in `request_irq`) is removed, but the interrupt line still remains, and will be disabled only when the last handler is removed. `free_irq` will block until any executing interrupts for the specified IRQ have completed. You must then avoid both `request_irq` and `free_irq` in the interrupt context.

```

ssize_t my_read(struct file *filp, char __user *buf, size_t count, loff_t
*f_pos)

{
    unsigned long flags; /* some stuff */

    [...]

    unsigned long flags; spin_lock_irqsave(&my_lock, flags); data++;
    spin_unlock_irqrestore(&my_lock, flags) [...]
}

```

```

static irqreturn_t my_interrupt_handler(int irq, void *p) {

    /*
     * preemption is disabled when running interrupt handler * also, the
     serviced irq line is disabled until the handler has completed * no need
     then to disable all other irq. We just use spin_lock and * spin_unlock
     */

    spin_lock(&my_lock); /* process data */

    [...]

    spin_unlock(&my_lock); return IRQ_HANDLED; }

```

When sharing data between different interrupt handlers (that is, the same driver managing two or more devices, each having its own IRQ

line), one should also protect that data with `spin_lock_irqsave()` in those handlers, in order to prevent the other IRQs from being triggered and uselessly spinning.

Concept of bottom halves

Bottom halves are mechanisms by which you split interrupt handlers into two part. This introduces another term, which is top half. Before discussing each of them, let us talk about their origin, and what problem they solve.

The problem – interrupt handler design limitations

Whether an interrupt handler holds a spinlock or not, preemption is disabled on the CPU running that handler. The more one wastes time in the handler, the less CPU is granted to the other task, which may considerably increase latency of other interrupts and so increase the latency of the whole system. The challenge is to acknowledge the device that raised the interrupt as quickly as possible in order to keep the system responsive.

On Linux systems (actually on all OS, by hardware design), any interrupt handler runs with its current interrupt line disabled on all processors, and sometimes you may need to disable all interrupts on the CPU actually running the handler, but you definitely don't want to miss an interrupt. To meet this need, the concept of *halves* has been introduced.

The solution – bottom halves

This idea consists of splitting the handler into two parts:

- The first part, called the top half or hard-IRQ, which is the registered function using `request_irq()` that will eventually mask/hide interrupts (on the current CPU, except the one being serviced since it is already disabled by the kernel before running the handler) depending on the needs, performs quick and fast operations (essentially time-sensitive tasks, read/write hardware registers, and fast processing of this data), schedules the second and next part, and then acknowledges the line. All interrupts that are disabled must have been re-enabled just before exiting the bottom half.
- The second part, called the bottom half, will process time-consuming stuff, and run with interrupt re-enabled. This way, you have the chance not to miss an interrupt.

Bottom halves are designed using a work-deferring mechanism, which we have seen previously. Depending on which one you choose, it may run in a (software) interrupt context, or in a process context. Bottom halves' mechanisms are:

- Softirqs
- Tasklets
- Workqueues
- Threaded IRQs

Softirqs and tasklets execute in a (software) interrupt context (meaning that preemption is disabled), Workqueues and threaded IRQs are executed in a process (or simply task) context, and can be preempted, but nothing prevents us from changing their real-time properties to fit your needs and change their preemption behavior (see `CONFIG_PREEMPT` OR `CONFIG_PREEMPT_VOLUNTARY`. This also impacts the whole system). Bottom halves are not always possible. But when it is possible, it is certainly the best thing to do.

grep -rn tasklet_schedule

```
struct my_data {  
    int my_int_var; struct tasklet_struct the_tasklet; int dma_request; };
```

```
static void my_tasklet_work(unsigned long data) {  
    /* Do what ever you want here */  
}
```

```
struct my_data *md = init_my_data;
```

```
/* somewhere in the probe or init function */
```

```
[...]
```

```
tasklet_init(&md->the_tasklet, my_tasklet_work, (unsigned  
long)md); [...]
```

```
static irqreturn_t my_irq_handler(int irq, void *dev_id) {
```

```
    struct my_data *md = dev_id;
```

```
    /* Let's schedule our tasklet */
```

```
tasklet_schedule(&md.dma_tasklet);
```

```
return IRQ_HANDLED; }
```

In the preceding sample, our tasklet will execute the function `my_tasklet_work()`.

```

static DECLARE_WAIT_QUEUE_HEAD(my_wq); /* declare and
init the wait queue */

static struct work_struct my_work;

/* some where in the probe function */

/*
* work queue initialization. "work_handler" is the call back that will
be * executed when our work is scheduled.
*/

INIT_WORK(my_work, work_handler);

static irqreturn_t my_interrupt_handler(int irq, void *dev_id) {

    uint32_t val; struct my_data = dev_id;

    val = readl(my_data->reg_base + REG_OFFSET); if (val ==
0xFFCD45EE) {

        my_data->done = true; wake_up_interruptible(&my_wq); } else {

        schedule_work(&my_work); }

    return IRQ_HANDLED; };

```

In the preceding sample, we used either a wait queue or a work queue in order to wake up a possibly sleeping process waiting for us, or schedule a work depending on the value of a register. We have no

shared data or resource, so there is no need to disable all other IRQs (spin_lock_irq_disable).

Softirqs as bottom half

As said in the beginning of this chapter, we will not discuss softirq. Tasklets will be enough everywhere you feel the need to use softirqs. Anyway, let's talk about their defaults.

Softirqs run in a software interrupt context, with preemption disabled, holding the CPU until they complete. Softirq should be fast; otherwise they may slow the system down. When, for any reason, a softirq prevents the kernel from scheduling other tasks, any new incoming softirq will be handled by **ksoftirqd** threads, running in a process context.

```

int request_threaded_irq(unsigned int irq, irq_handler_t handler,\
    irq_handler_t thread_fn, \
    unsigned long irqflags, \
    const char *devname, void *dev_id)

/*
 * Default primary interrupt handler for threaded interrupts. Is *
 * assigned as primary handler when request_threaded_irq is called *
 * with handler == NULL. Useful for oneshot interrupts.
 */

static irqreturn_t irq_default_primary_handler(int irq, void *dev_id) {

    return IRQ_WAKE_THREAD; }

request_threaded_irq(unsigned int irq, irq_handler_t handler,
    irq_handler_t thread_fn, unsigned long irqflags, const char *devname,
    void *dev_id) {

    [...]

    if (!handler) {

        if (!thread_fn) return -EINVAL; handler =
        irq_default_primary_handler; }

    [...]

```

```
}
```

```
EXPORT_SYMBOL(request_threaded_irq);
```

```
request_irq(unsigned int irq, irq_handler_t handler, \
```

```
    unsigned long flags, const char *name, void *dev) {
```

```
    return request_threaded_irq(irq, handler, NULL, flags, \
```

```
    name, dev);
```

```
}
```

```

static irqreturn_t pcf8574_kp_irq_handler(int irq, void *dev_id) {

    struct custom_data *lp = dev_id; unsigned char nextstate =
    read_state(lp);

    if (lp->laststate != nextstate) {

        int key_down = nextstate < ARRAY_SIZE(lp->btncode); unsigned
        short keycode = key_down ?

        lp->btncode[nextstate] : lp->btncode[lp->laststate];

        input_report_key(lp->idev, keycode, key_down); input_sync(lp-
        >idev); lp->laststate = nextstate; }

    return IRQ_HANDLED;

}

```

```

static int pcf8574_kp_probe(struct i2c_client *client, \

```

```

    const struct i2c_device_id *id) {

```

```

    struct custom_data *lp = init_custom_data(); [...]

```

```

    /*

```

```

    * @handler is NULL and @thread_fn != NULL

```

```

    * the default primary handler is installed, which will * return
    IRQ_WAKE_THREAD, that will schedule the thread * asociated to
    the bottom half. the bottom half must then * return IRQ_HANDLED

```

```

when finished */

ret = request_threaded_irq(client->irq, NULL, \
pcf8574_kp_irq_handler, \
IRQF_TRIGGER_LOW | IRQF_ONESHOT, \
DRV_NAME, lp);

if (ret) {

dev_err(&client->dev, "IRQ %d is not free\n", \
client->irq);

goto fail_free_device;

}

ret = input_register_device(idev); [...]

}

```

When an interrupt handler is executed, the serviced IRQ is always disabled on all CPUs, and re-enabled when the hard-IRQ (top-half) finishes. But if for any reason you need the IRQ line not to be re-enabled after the top half, and to remain disabled until the threaded handler has been run, you should request the threaded IRQ with the flag `IRQF_ONESHOT` enabled (by just doing an OR operation as shown previously). The IRQ line will then be re-enabled after the bottom half has finished.

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

```
#include <linux/module.h> #include <linux/workqueue.h> /* for work  
queue */
```

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

```
static struct delayed_work initiate_shutdown_work; static void  
delayed_shutdown( void ) {
```

```
    char *cmd = "/sbin/shutdown"; char *argv[] = {
```

```
    cmd,
```

```
    "-h",
```

```
    "now",
```

```
    NULL,
```

```
};
```

```
char *envp[] = {
```

```
    "HOME=/", "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL,
```

```
};
```

```
    call_usermodehelper(cmd, argv, envp, 0); }
```

```

static int __init my_shutdown_init( void ) {

    schedule_delayed_work(&delayed_shutdown,
msecs_to_jiffies(200)); return 0; }


static void __exit my_shutdown_exit( void ) {

    return; }


module_init( my_shutdown_init ); module_exit( my_shutdown_exit );

MODULE_LICENSE("GPL");

MODULE_AUTHOR("John Madieu", <john.madieu@gmail.com>);
MODULE_DESCRIPTION("Simple module that trigger a delayed
shut down");

```

In the preceding example, the API used (`call_usermodehelper`) is a part of the Usermode-helper API, with all functions defined in `kernel/kmod.c`. Its use is quite simple; just a look inside `kmod.c` will give you an idea. You may be wondering what this API was defined for. It is used by the kernel, for example, for module (un)loading and cgroups management.

Summary

In this chapter, we discussed about the fundamental elements to start driver development, presenting every mechanism frequently used in drivers. This chapter is very important, since it discusses topics other chapters in this book rely on. The next chapter for example, dealing with character devices, will use some of elements discussed in this chapter.


```
struct cdev {
```

```
    struct kobject kobj; struct module *owner; const struct  
    file_operations *ops; struct list_head list; dev_t dev; unsigned int  
    count; };
```

This chapter will walk through the specificities of character device drivers, explain how they create, identify, and register the devices with the system, and also give a better overview of the device file methods, which are methods by which the kernel exposes the device capabilities to user space, accessible by using file related system calls (read, write, select, open, close and so on), described in struct file_operations structures, which you have certainly heard of before.

The concept behind major and minor

Character devices are populated in the `/dev` directory. Do note that, they are not only files present in that directory. A character device file is recognizable to its type, which we can display thanks to the command `ls -l`. Major and minor identify and tie the devices with the drivers. Let us see how it works, by listing the content of the `/dev` directory (`ls -l /dev`):

```
[...]
drwxr-xr-x 2 root root 160 Mar 21 08:57 input
crw-r----- 1 root kmem 1, 2 Mar 21 08:57 kmem
lrwxrwxrwx 1 root root 28 Mar 21 08:57 log -> /run/systemd/journal/dev-log
crw-rw---- 1 root disk 10, 237 Mar 21 08:57 loop-control
brw-rw---- 1 root disk 7, 0 Mar 21 08:57 loop0
brw-rw---- 1 root disk 7, 1 Mar 21 08:57 loop1
brw-rw---- 1 root disk 7, 2 Mar 21 08:57 loop2
brw-rw---- 1 root disk 7, 3 Mar 21 08:57 loop3
```

Given the preceding excerpt, the first character of the first column identifies the file type. Possible values are:

- `c`: This is for character device files
- `b`: This is for block device file
- `l`: This is for symbolic link
- `d`: This is for directory
- `s`: This is for socket
- `p`: This is for named pipe

For `b` and `c` file types, the fifth and sixth columns right before the date respect the `<x, y>` pattern. `x` represents the major, and `y` is the minor. For example, the third line is `<1, 2>` and the last one is `<7, 3>`. That is one of the classical methods for identifying a character device file from user space, as well as its major and minor.

The kernel holds the numbers that identify a device in `dev_t` type variables, which are simply `u32` (32-bit unsigned long). The major is represented with only 12 bits, whereas the minor is coded on the 20 remaining bits.

As one can see in `include/linux/kdev_t.h`, given a `dev_t` type variable, one may need to extract the minor or the major. The kernel provides a macro for these

purposes:

```
| MAJOR(dev_t dev);  
| MINOR(dev_t dev);
```

On the other hand, you may have a minor and a major, and need to build a `dev_t`. The macro you should use is `MKDEV(int major, int minor);`:

```
| #define MINORBITS    20  
| #define MINORMASK    ((1U << MINORBITS) - 1)  
| #define MAJOR(dev)   ((unsigned int) ((dev) >> MINORBITS))  
| #define MINOR(dev)   ((unsigned int) ((dev) & MINORMASK))  
| #define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

The device is registered with a major number that identifies the device, and a minor, which one may use as an array index to a local list of devices, since one instance of the same driver may handle several devices while different drivers may handle different devices of the same type.

Device number allocation and freeing

Device numbers identify device files across the system. That means, there are two ways to allocate these device numbers (actually major and minor):

- **Statically:** Guessing a major not yet used by another driver using the `register_chrdev_region()` function. One should avoid using this as much as possible. Its prototype looks this:

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

This method returns 0 on success, or a negative error code on failure. `first` is made of the major number that we need along with the first minor of the desired range. One should use `MKDEV(ma,mi)`. `count` is the number of consecutive device numbers required, and `name` should be the name of the associated device or driver.

- **Dynamically:** Letting the kernel do the job for us, using the `alloc_chrdev_region()` function. This is the recommended way to obtain a valid device number. Its prototype is as follows:

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

This method returns 0 on success, or a negative error code on failure. `dev` is the only output parameter. It represents the first number the kernel assigned. `firstminor` is the first of the requested range of minor numbers, `count` the number of minors one requires, and `name` should be the name of the associated device or driver.

The difference between the two is that with the former, one should know in advance what number we need. This is registration: one tells the kernel what device numbers we want. This may be used for pedagogic purposes, and works as long as the only user of the driver is you. When it comes to loading the driver on another machine, there is no guarantee the chosen number is free on that machine, and this will lead to conflicts and trouble. The second method is cleaner and much safer, since the kernel is responsible for guessing the right

numbers for us. We do not even have to care about what the behavior would be on loading the module on to another machine, since the kernel will adapt accordingly.

Anyway, the preceding functions are generally not called directly from the driver, but masked by the framework on which the driver relies (IIO framework, input framework, RTC, and so on), by means of dedicated API. These frameworks are all discussed in further chapters in the book.

```

struct file_operations {

    struct module *owner;

    loff_t (*llseek) (struct file *, loff_t, int); ssize_t (*read) (struct file
*, char __user *, size_t, loff_t *); ssize_t (*write) (struct file *, const
char __user *, size_t, loff_t *); unsigned int (*poll) (struct file *,
struct poll_table_struct *); int (*mmap) (struct file *, struct
vm_area_struct *); int (*open) (struct inode *, struct file *); long
(*unlocked_ioctl) (struct file *, unsigned int, unsigned long); int
(*release) (struct inode *, struct file *); int (*fsync) (struct file *,
loff_t, loff_t, int datasync); int (*fasync) (int, struct file *, int); int
(*lock) (struct file *, int, struct file_lock *); int (*flock) (struct file *,
int, struct file_lock *); [...]

};

```

The preceding excerpt only lists important methods of the structure, especially the ones that are relevant for the needs of this book. One can find the full description in `include/linux/fs.h` in kernel sources. Each of these callbacks is linked with a system call, and none of them is mandatory. When a user code calls a files-related system call on a given file, the kernel looks for the driver responsible for that file (especially the one that created the 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 it. If not, it returns an error code that varies depending on the system call. For example, an undefined `(*mmap)` method will return `-ENODEV` to user, whereas an undefined `(*write)` method will return `-EINVAL`.

```

struct inode {

    [...]

    struct pipe_inode_info *i_pipe; /* Set and used if this is a *linux
kernel pipe */

    struct block_device *i_bdev; /* Set and used if this is a * a block
device */

    struct cdev *i_cdev; /* Set and used if this is a * character device */

    [...]

}

struct file {

    [...]

    struct path f_path; /* Path to the file */

    struct inode *f_inode; /* inode associated to this file */

    const struct file_operations *f_op; /* operations that can be *
performed on this file */

    loff_t f_pos; /* Position of the cursor in * this file */

    /* needed for tty driver, and maybe others */

    void *private_data; /* private data that driver can set * in order to
share some data between file * operations. This can point to any data
* structure.

```

```
*/
```

```
[...]
```

```
}
```

The difference between `struct inode` and `struct file` is that an `inode` doesn't track the current position within the file or the current mode. It only contains stuff that helps the OS find the contents of the underlying file structure (pipe, directory, regular disk file, block/character device file, and so on). On the other hand, the `struct file` is used as a generic structure (it actually holds a pointer to a `struct inode` structure) that represents an open file and provides a set of functions related to methods one can perform on the underlying file structure. Such methods are: `open`, `write`, `seek`, `read`, `select`, and so on. All this reinforces the philosophy of UNIX systems that says *everything is file*.

In other words, a `struct inode` represents a file in the kernel, and a `struct file` describes it when it is actually open. There may be different file descriptors that represent the same file opened several times, but these will point to the same `inode`.


```
#define EEP_NBANK 8

#define EEP_DEVICE_NAME "eep-mem"

#define EEP_CLASS "eep-class"


struct class *eep_class;

struct cdev eep_cdev[EEP_NBANK];

dev_t dev_num;


static int __init my_init(void)
{
    int i;

    dev_t curr_dev;


    /* Request the kernel for EEP_NBANK devices */

    alloc_chrdev_region(&dev_num, 0, EEP_NBANK,
EEP_DEVICE_NAME);

    /* Let's create our device's class, visible in /sys/class */

    eep_class = class_create(THIS_MODULE, EEP_CLASS);
```

```

/* Each eeprom bank represented as a char device (cdev) */

for (i = 0; i < EEP_NBANK; i++) {

    /* Tie file_operations to the cdev */

    cdev_init(&my_cdev[i], &eep_fops); eep_cdev[i].owner =
THIS_MODULE;

    /* Device number to use to add cdev to the core */

    curr_dev = MKDEV(MAJOR(dev_num), MINOR(dev_num) + i);

    /* Now make the device live for the users to access */

    cdev_add(&eep_cdev[i], curr_dev, 1);

    /* create a device node each device /dev/eep-mem0, /dev/eep-
mem1, * With our class used here, devices can also be viewed under *
/sys/class/eep-class.

    */

    device_create(eep_class, NULL, /* no parent device */

    curr_dev,

    NULL, /* no additional data */

    EEP_DEVICE_NAME "%d", i); /* eep-mem[0-7] */

}

return 0;

```

}

Writing file operations

After introducing the preceding file operations, it is time to implement them in order to enhance the driver capabilities and expose the device's methods to the user space (by means of system calls or course). Each of these methods has its particularities, which we will highlight in this section.

Exchanging data between kernel space and user space

This section does not describe any driver file operation but instead, introduces some kernel facilities that one may use to write these driver methods. The driver's `write()` method consists of reading data from user space to kernel space, and then processing that data from the kernel. Such processing could be something like *pushing* the data to the device, for example. On the other hand, the driver's `read()` method consists of copying data from the kernel to the user space. Both of these methods introduces new elements we need to discuss prior to jumping to their respective steps. The first one is `__user`. `__user` is a cookie used by sparse (a semantic checker used by the kernel to find possible coding faults) to let the developer know he is actually about to use an untrusted pointer (or a pointer that may be invalid in the current virtual address mapping) improperly and that he should not dereference but instead, use dedicated kernel functions to access the memory to which this pointer points.

This allows us to introduce different kernel functions needed to access such memory, either to read or write. These are `copy_from_user()` and `copy_to_user()` respectively to copy a buffer from user space to kernel space, and vice versa, to copy a buffer from kernel to user space:

```
unsigned long copy_from_user(void *to, const void __user *from,
                             unsigned long n)
unsigned long copy_to_user(void __user *to, const void *from,
                          unsigned long n)
```

In both cases, pointers prefixed with `__user` point to user space (untrusted) memory. `n` represents the number of bytes to copy. `from` represents the source address, and `to` is the destination address. Each of these returns the number of bytes that could not be copied. On success, the return value should be `0`.

Please do note that with `copy_to_user()`, if some data could not be copied, the function will pad the copied data to the requested size using zero bytes.

A single value copy

When it comes to copying single and simple variables like `char` and `int` but not larger data types like structures nor arrays, the kernel offers dedicated macros in order to quickly perform the desired operation. These macros are `put_user(x, ptr)` and `get_user(x, ptr)`, which are explained as follows:

- `put_user(x, ptr);`: This macro copies a variable from kernel space to user space. `x` represents value to copy to user space, and `ptr` is the destination address in user space. The macro returns `0` on success, or `-EFAULT` on error. `x` must be assignable to the result of dereferencing `ptr`. In other words, they must have (or point to) the same type.
- `get_user(x, ptr);`: This macro copies a variable from user space to kernel space, and returns `0` on success or `-EFAULT` on error. Please do note that `x` is set to `0` on error. `x` represents the kernel variable to store the result, and `ptr` is the source address in user space. The result of dereferencing `ptr` must be assignable to `x` without a cast. Guess what it means.

```
int (*open)(struct inode *inode, struct file *filp);
```

```

struct pcf2127 {

    struct cdev cdev; unsigned char *sram_data; struct i2c_client
    *client; int sram_size; [...]

};

static unsigned int sram_major = 0;

static struct class *sram_class = NULL;

static int sram_open(struct inode *inode, struct file *filp) {

    unsigned int maj = imajor(inode); unsigned int min = iminor(inode);

    struct pcf2127 *pcf = NULL; pcf = container_of(inode->i_cdev,
    struct pcf2127, cdev); pcf->sram_size = SRAM_SIZE;

    if (maj != sram_major || min < 0 ){

        pr_err ("device not found\n"); return -ENODEV; /* No such device
        */

    }

    /* prepare the buffer if the device is opened for the first time */

    if (pcf->sram_data == NULL) {

        pcf->sram_data = kzalloc(pcf->sram_size, GFP_KERNEL); if (pcf-
        >sram_data == NULL) {

            pr_err("Open: memory allocation failed\n"); return -ENOMEM; }

```



```
}
```

```
filp->private_data = pcf; return 0;
```

```
}
```

```

static int sram_release(struct inode *inode, struct file *filp) {

    struct pcf2127 *pcf = NULL; pcf = container_of(inode->i_cdev,
struct pcf2127, cdev);

    mutex_lock(&device_list_lock); filp->private_data = NULL;

    /* last close? */

    pcf2127->users--; if (!pcf2127->users) {

        kfree(tx_buffer); kfree(rx_buffer); tx_buffer = NULL; rx_buffer =
NULL;

        [...]

        if (any_global_struct) kfree(any_global_struct); }

    mutex_unlock(&device_list_lock);

    return 0;

}

```

```
ssize_t(*write)(struct file *filp, const char __user *buf, size_t count,  
loff_t *pos);
```

- The return value is the number of bytes (size) written
- *buf represents the data buffer coming from the user space
- count is the size of the requested transfer
- *pos indicates the start position from which data should be written in the file

ssize_t

```
EEPROM_WRITE(struct file *filp, const char __user *buf, size_t count,  
loff_t *f_pos)
```

```
{
```

```
    struct eeprom_dev *eep = filp->private_data; ssize_t retval = 0;
```

```
    /* step (1) */
```

```
    if (*f_pos >= eep->part_size) /* Writing beyond the end of a  
partition is not allowed. */
```

```
        return -EINVAL;
```

```
    /* step (2) */
```

```
    if (*pos + count > eep->part_size) count = eep->part_size - *pos;
```

```
    /* step (3) */
```

```
    int part_origin = PART_SIZE * eep->part_index; int  
    register_address = part_origin + *pos;
```

```
    /* step(4) */
```

```
    /* Copy data from user space to kernel space */
```

```
    if (copy_from_user(eep->data, buf, count) != 0) return -EFAULT;
```

```
    /* step (5) */
```

```
    /* perform the write to the device */
```

```
if (write_to_device(register_address, buff, count) < 0){  
    pr_err("ee24lc512: i2c_transfer failed\n"); return -EFAULT; }  
  
/* step (6) */  
*f_pos += count; return count;  
}
```

```
ssize_t (*read) (struct file *filp, char __user *buf, size_t count, loff_t *pos);
```

The return value is the size read. The rest of the method's elements are described here:

- *buf is the buffer we receive from the user space
- count is the size of the requested transfer (size of the user buffer)
- *pos indicates the start position from which data should be read in the file

```

ssize_t eep_read(struct file *filp, char __user *buf, size_t count, loff_t
*f_pos) {

    struct eeprom_dev *eep = filp->private_data;

    if (*f_pos >= EEP_SIZE) /* EOF */

    return 0;


    if (*f_pos + count > EEP_SIZE) count = EEP_SIZE - *f_pos;

    /* Find location of next data bytes */

    int part_origin = PART_SIZE * eep->part_index; int
eep_reg_addr_start = part_origin + *pos;

    /* perform the read from the device */

    if (read_from_device(eep_reg_addr_start, buff, count) < 0){

    pr_err("ee24lc512: i2c_transfer failed\n"); return -EFAULT; }


    /* copy from kernel to user space */

    if(copy_to_user(buf, dev->data, count) != 0) return -EIO;


    *f_pos += count; return count;

}

```

`loff_t(*llseek) (structfile *filp, loff_t offset, int whence);`

- The return value is the new position in the file
- `loff_t` is an offset, relative to the current file position, which defines how much it will be changed
- `whence` defines where to seek from. Possible values are:
 - `SEEK_SET`: This puts the cursor into a position relative to the beginning of the file
 - `SEEK_CUR`: This puts the cursor into a position relative to the current file position
 - `SEEK_END`: This adjusts the cursor to a position relative to end-of-file


```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
#include <sys/types.h>
```

```
#include <stdio.h>
```

```
#define CHAR_DEVICE "toto.txt"
```

```
int main(int argc, char **argv) {
```

```
    int fd= 0; char buf[20];
```

```
    if ((fd = open(CHAR_DEVICE, O_RDONLY)) < -1) return 1;
```

```
    /* Read 20 bytes */
```

```
    if (read(fd, buf, 20) != 20) return 1; printf("%s\n", buf);
```

```
    /* Move the cursor to 10 time, relative to its actual position */
```

```
    if (lseek(fd, 10, SEEK_CUR) < 0) return 1; if (read(fd, buf, 20) !=  
20) return 1; printf("%s\n",buf);
```

```
    /* Move the cursor ten time, relative from the beginig of the file */
```

```
    if (lseek(fd, 7, SEEK_SET) < 0) return 1; if (read(fd, buf, 20) != 20)  
return 1; printf("%s\n",buf);
```

```
    close(fd); return 0; }
```

jma@jma:~/work/tutos/sources\$ cat toto.txt

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua.

jma@jma:~/work/tutos/sources\$
./seek
Lorem ipsum dolor si

nsectetur adipiscing
psum dolor sit
amet,

jma@jma:~/work/tutos/sources\$

The poll method

If one needs to implement a passive wait (not wasting CPU cycles while sensing the character device), one must implement the `poll()` function, which will be called whenever a user-space program performs a `select()` or `poll()` system calls on the file associated with the device:

```
| unsigned int (*poll) (struct file *, struct poll_table_struct *);
```

The kernel function at the heart of this method is `poll_wait()`, defined in `<linux/poll.h>`, which is the header one should include in driver code:

```
| void poll_wait(struct file * filp, wait_queue_head_t * wait_address,  
| poll_table *p)
```

`poll_wait()` adds the device associated with a `struct file` structure (given as first parameter) to a list of those that can wake up processes (which have been put to sleep in the `struct wait_queue_head_t` structure given as second parameter), according to events registered into the `struct poll_table` structure given as third parameter. A user process can run `poll()`, `select()`, or `epoll()` system calls to add a set of files to a list on which it needs to wait, in order to be aware of the associated (if any) devices readiness. The kernel will then call the `poll` entry of the driver associated with each device file. The `poll` method of each driver should then call `poll_wait()` in order to register events for which the process needs to be notified with the kernel, put that process to sleep until one of these events occurs, and register the driver as one of those that can wake the process up. The usual way is to use a wait queue per event type (one for readability, another one for writability, and eventually one for exception if needed), according to events supported by the `select()` (or `poll()`) system call.

The return value of the `(*poll)` file operation must have `POLLIN | POLLRDNORM` set if there is data to read (at the moment, `select` or `poll` is called), `POLLOUT | POLLWRNORM` if the device is writable (at the moment, `select` or `poll` is called here as well), and `0` if there is no new data and the device is not yet writable. In the following example, we assume the device supports both blocking read and write. Of course one may implement only one of these. If the driver does not define this method, the device will be considered as always readable and writable, so that `poll()` or

`select()` system calls return immediately.

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <sys/select.h>
```

```
#define NUMBER_OF_BYTE 100
```

```
#define CHAR_DEVICE "/dev/packt_char"
```

```
char data[NUMBER_OF_BYTE];
```

```
int main(int argc, char **argv)
```

```
{
```

```
    int fd, retval; ssize_t read_count; fd_set readfds;
```

```
    fd = open(CHAR_DEVICE, O_RDONLY); if(fd < 0) /* Print a  
message and exit*/
```

```
    [...]
```

```

while(1){

FD_ZERO(&readfds); FD_SET(fd, &readfds);

/*

* One needs to be notified of "read" events only, without timeout.

* This call will put the process to sleep until it is notified the * event
for which it registered itself */

ret = select(fd + 1, &readfds, NULL, NULL, NULL);

/* From this line, the process has been notified already */

if (ret == -1) {

    fprintf(stderr, "select call on %s: an error occurred",
CHAR_DEVICE); break;

}

/*

* file descriptor is now ready.

* This step assume we are interested in one file only.

*/

if (FD_ISSET(fd, &readfds)) {

    read_count = read(fd, data, NUMBER_OF_BYTE); if (read_count
< 0 ) /* An error occurred. Handle this */

```

[...]

if (read_count != NUMBER_OF_BYTE) /* We have read less than
need bytes */

[...] /* handle this */

else

/* Now we can process data we have read */

[...]

}

}

close(fd);

return EXIT_SUCCESS; }

The ioctl method

A typical Linux system contains around 350 system calls (syscalls), but only a few of them are linked with file operations. Sometimes devices may need to implement specific commands that are not provided by system calls, and especially the ones associated with files and thus device files. In this case, the solution is to use **input/output control(ioctl)**, which is a method by which one extends a list of syscalls (actually commands) associated with a device.. One can use it to send special commands to devices (`reset`, `shutdown`, `configure`, and so on). If the driver does not define this method, the kernel will return `-ENOTTY` error to any `ioctl()` system call.

In order to be valid and safe, an `ioctl` command needs to be identified by a number which should be unique to the system. The unicity of `ioctl` numbers across the system will prevent it from sending the right command to the wrong device, or passing the wrong argument to the right command (given a duplicated `ioctl` number). Linux provides four helper macros to create an `ioctl` identifier, depending on whether there is data transfer or not and on the direction of the transfer. Their respective prototypes are:

```
|_IO(MAGIC, SEQ_NO)  
|_IOW(MAGIC, SEQ_NO, TYPE)  
|_IOR(MAGIC, SEQ_NO, TYPE)  
|_IOWR(MAGIC, SEQ_NO, TYPE)
```

Their descriptions are as follows:

- `_IO`: The `ioctl` does not need data transfer
- `_IOW`: The `ioctl` needs write parameters (`copy_from_user` OR `get_user`)
- `_IOR`: The `ioctl` needs read parameters (`copy_to_user` OR `put_user`)
- `_IOWR`: The `ioctl` needs both write and read parameters

What their parameters mean (in the order they are passed) is described here:

1. A number coded on 8 bits (0 to 255), called magic number.
2. A sequence number or command ID, also on 8 bits.
3. A data type, if any, that will inform the kernel about the size to be copied.

It is well documented in *Documentation/ioctl/ioctl-decoding.txt* in the kernel source, and existing `ioctl` are listed in *Documentation/ioctl/ioctl-number.txt*, a good place to start when you need to create an `ioctl` command.

```
#ifndef PACKET_IOCTL_H
```

```
#define PACKET_IOCTL_H
```

```
/*
```

```
* We need to choose a magic number for our driver, and sequential  
numbers * for each command: */
```

```
#define EEP_MAGIC 'E'
```

```
#define ERASE_SEQ_NO 0x01
```

```
#define RENAME_SEQ_NO 0x02
```

```
#define CLEAR_BYTE_SEQ_NO 0x03
```

```
#define GET_SIZE 0x04
```

```
/*
```

```
* Partition name must be 32 byte max */
```

```
#define MAX_PART_NAME 32
```

```
/*
```

```
* Now let's define our ioctl numbers: */
```

```
#define EEP_ERASE _IO(EEP_MAGIC, ERASE_SEQ_NO) #define  
EEP_RENAME_PART _IOW(EEP_MAGIC, RENAME_SEQ_NO,
```

```
unsigned long) #define EEP_GET_SIZE _IOR(EEP_MAGIC,  
GET_SIZE, int *) #endif
```

```

long ioctl(struct file *f, unsigned int cmd, unsigned long arg);

/*
 * User space code also need to include the header file in which ioctls
 * defined are defined. This is eep_ioctl.h in our case.
 */

#include "eep_ioctl.h"

static long eep_ioctl(struct file *f, unsigned int cmd, unsigned long
arg) {

    int part; char *buf = NULL; int size = 1300;

    switch(cmd){

        case EEP_ERASE: erase_eepreom(); break;

        case EEP_RENAME_PART: buf = kmalloc(MAX_PART_NAME,
GFP_KERNEL); copy_from_user(buf, (char *)arg,
MAX_PART_NAME); rename_part(buf); break;

        case EEP_GET_SIZE: copy_to_user((int*)arg, &size, sizeof(int));
break;

        default: return -ENOTTY; }

    return 0; }

#include <stdio.h>

#include <stdlib.h>

```

```
#include <fcntl.h>

#include <unistd.h>

#include "eep_ioctl.h" /* our ioctl header file */


int main()

{

    int size = 0; int fd;

    char *new_name = "lorem_ipsum"; /* must not be longer than
MAX_PART_NAME */


    fd = open("/dev/eep-mem1", O_RDWR); if (fd == -1){
printf("Error while opening the eeprom\n"); return -1; }


    ioctl(fd, EEP_ERASE); /* ioctl call to erase partition */

    ioctl(fd, EEP_GET_SIZE, &size); /* ioctl call to get partition size
*/

    ioctl(fd, EEP_RENAME_PART, new_name); /* ioctl call to rename
partition */


    close(fd); return 0; }
```

```
static const struct file_operations eep_fops = {  
  
    .owner = THIS_MODULE, .read = eep_read, .write = eep_write,  
    .open = eep_open, .release = eep_release, .llseek = eep_llseek, .poll =  
    eep_poll, .unlocked_ioctl = eep_ioctl, };
```

Let us remember, the structure is given as a parameter to `cdev_init` in the `init` method.

Summary

In this chapter, we have demystified character devices and we have seen how to let users interact with our driver through device files. We learned how to expose file operations to the user space and control their behavior from within the kernel. We went so far that you are even able to implement multi-device support. The next chapter is a bit hardware oriented since it deals with platform drivers which expose hardware device capabilities to the user space. The power of character drivers combined with platform drivers is just amazing. See you in the next chapter.

Platform Device Drivers

We all know about plug and play devices. They are handled by the kernel as soon as they are plugged in. These may be USB or PCI express, or any other auto-discovered devices. Therefore, other device types also exist, which are not hot-pluggable, and which the kernel needs to know about prior to being managed. There are I2C, UART, SPI, and other devices not wired to enumeration-capable buses.

There are real physical buses you may already know: USB, I2S, I2C, UART, SPI, PCI, SATA, and so on. Such buses are hardware devices named controllers. Since they are a part of SoC, they can't be removed, are non-discoverable, and are also called platform devices.



People often say platform devices are on-chip devices (embedded in the SoC). In practice, it is partially true, since they are hard-wired into the chip and can't be removed. But devices connected to I2C or SPI are not on-chip, and are platform devices too, because they are not discoverable. Similarly, there may be on-chip PCI or USB devices, but they are not platform devices, because they are discoverable.

From an SoC point of view, those devices (buses) are connected internally through dedicated buses, and are most of the time proprietary and specific to the manufacturer. From the kernel point of view, these are root devices, and connected to nothing. That is where the *pseudo platform bus* comes in. The pseudo platform bus, also called platform bus is a kernel virtual bus for devices that do not seat on a physical bus known to the kernel. In this chapter, platform devices refer to devices that rely on the pseudo platform bus.

Dealing with platform devices essentially requires two steps:

- Register a platform driver (with a unique name) that will manage your devices
- Register your platform device with the same name as the driver, and their resources, in order to let the kernel know that your device is there

That being said, in this chapter, we will discuss the following:

- Platform devices along with their driver
- Devices and driver-matching mechanisms in the kernel
- Registering platform drivers with devices, as well as platform data

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

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

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

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

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

```
static int my_pdrv_probe (struct platform_device *pdev){  
    pr_info("Hello! device probed!\n"); return 0;  
}
```

```
static void my_pdrv_remove(struct platform_device *pdev){  
    pr_info("good bye reader!\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_drv_init(void) {  
    pr_info("Hello Guy\n");  
    /* Registering with Kernel */  
    platform_driver_register(&mypdrv); return 0;  
}
```

```
static void __exit my_pdrv_remove (void) {  
    Pr_info("Good bye Guy\n");  
    /* Unregistering from Kernel */  
    platform_driver_unregister(&my_driver); }  
  
module_init(my_drv_init);  
  
module_exit(my_pdrv_remove);
```

```
<span class="msoIns">MODULE_LICENSE(</span>"<span  
class="msoIns">GPL</span>"<span class="msoIns">);<br/></span>  
<span class="msoIns">MODULE_AUTHOR(</span>"<span  
class="msoIns">John Madieu</span>"<span class="msoIns">);<br/>
```

```
</span><span class="msoIns">MODULE_DESCRIPTION(</span>"  
<span class="msoIns">My platform Hello World module</span>"  
<span class="msoIns">);</span>
```

```
/*
```

```
* module_platform_driver() - Helper macro for drivers that don't * do  
anything special in module init/exit. This eliminates a lot * of  
boilerplate. Each module may only use this macro once, and * calling  
it replaces module_init() and module_exit() */
```

```
#define module_platform_driver(__platform_driver) \
```

```
module_driver(__platform_driver, platform_driver_register, \  
platform_driver_unregister)
```

```
[...]
```

```
static int my_driver_probe (struct platform_device *pdev){
```

```
    [...]
```

```
}
```

```
static void my_driver_remove(struct platform_device *pdev){
```

```
    [...]
```

```
}
```

```
static struct platform_driver my_driver = {  
  
    [...]  
  
};  
  
module_platform_driver(my_driver);
```

There are specific macros for each bus that one needs to register the driver with. The following list is not exhaustive:

- `module_platform_driver(struct platform_driver)` for platform drivers, dedicated to devices that do not sit on conventional physical buses (we just used it above)
- `module_spi_driver(struct spi_driver)` for SPI drivers
- `module_i2c_driver(struct i2c_driver)` for I2C drivers
- `module_pci_driver(struct pci_driver)` for PCI drivers
- `module_usb_driver(struct usb_driver)` for USB drivers
- `module_mdio_driver(struct mdio_driver)` for mdio
- [...]

If you don't know which bus your driver needs to sit on, then it is a platform driver, and you should use `platform_driver_register` or `platform_driver_probe` to register the driver.

```
struct platform_device {  
    const char *name; u32 id; struct device dev; u32 num_resources;  
    struct resource *resource; };
```

When it comes to the platform driver, before driver and device match, the name field of both struct platform_device and static struct platform_driver.driver.name must be the same. The num_resources and struct resource *resource field will be covered in the next section. Just remember that, since resource is an array, num_resources must contain the size of that array.

Resources and platform data

At the opposite end to hot-pluggable devices, the kernel has no idea of what devices are present on your system, what they are capable of, or what they need in order to work properly. There is no auto-negotiation process, so any information provided to the kernel would be welcome. There are two methods to inform the kernel about the resources (irq, dma, memory region, I/O ports, buses) and data (any custom and private data structure you may want to pass to the driver) that the device needs which are discussed as follows:

Device provisioning - the old and depreciated way

This method is to be used with the kernel version that does not support device tree. With this method, drivers remain generic, and devices are registered in board-related source files.


```

#define IORESOURCE_IO 0x00000100 /* PCI/ISA I/O ports */

#define IORESOURCE_MEM 0x00000200 /* Memory regions */

#define IORESOURCE_REG 0x00000300 /* Register offsets */

#define IORESOURCE_IRQ 0x00000400 /* IRQ line */

#define IORESOURCE_DMA 0x00000800 /* DMA channels */

#define IORESOURCE_BUS 0x00001000 /* Bus */

struct resource {

    resource_size_t start; resource_size_t end; const char *name;
    unsigned long flags; };

int probe(struct platform_device *pdev);

struct resource *platform_get_resource(struct platform_device *dev,
    unsigned int type, unsigned int num);

static int my_driver_probe(struct platform_device *pdev) {

    struct my_gpios *my_gpio_pdata =

        (struct my_gpios*)dev_get_platdata(&pdev->dev);

    int rgpio = my_gpio_pdata->reset_gpio; int lgpio = my_gpio_pdata-
        >led_gpio;

    struct resource *res1, *res2; void *reg1, *reg2; int irqnum;

```

```

    res1 = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if(!res1){

        pr_err(" First Resource not available"); return -1;

    }

    res2 = platform_get_resource(pdev, IORESOURCE_MEM, 1);
    if(!res2){

        pr_err(" Second Resource not available"); return -1;

    }


    /* extract the irq */

    irqnum = platform_get_irq(pdev, 0); Pr_info("\n IRQ number of
Device: %d\n", irqnum);

    /*

    * At this step, we can use gpio_request, on gpio, * request_irq on
irqnum and ioremap() on reg1 and reg2.

    * ioremap() is discussed in chapter 11, Kernel Memory
Management */

    [...]

    return 0;

}

```

```

/*

* Other data than irq or memory must be embedded in a structure *
and passed to "platform_device.device.platform_data"

*/

struct my_gpios {

    int reset_gpio; int led_gpio;

};

/*our platform data*/

static struct my_gpiosneeded_gpios = {

    .reset_gpio = 47, .led_gpio = 41, };

/* Our resource array */

static struct resource needed_resources[] = {

    [0] = { /* The first memory region */

        .start = JZ4740_UDC_BASE_ADDR, .end =
JZ4740_UDC_BASE_ADDR + 0x10000 - 1, .flags =
IORESOURCE_MEM, .name = "mem1", },

    [1] = {

```

```
.start = JZ4740_UDC_BASE_ADDR2, .end =  
JZ4740_UDC_BASE_ADDR2 + 0x10000 -1, .flags =  
IORESOURCE_MEM, .name = "mem2", },
```

```
[2] = {
```

```
.start = JZ4740_IRQ_UDC, .end = JZ4740_IRQ_UDC, .flags =  
IORESOURCE_IRQ, .name = "mc",
```

```
},
```

```
};
```

```
static struct platform_devicemy_device = {
```

```
.name = "my-platform-device", .id = 0,
```

```
.dev = {
```

```
.platform_data = &needed_gpios, },
```

```
.resource = needed_resources, .num_resources =  
ARRAY_SIZE(needed_resources), };
```

```
platform_device_register(&my_device);
```

```
void *dev_get_platdata(const struct device *dev) struct my_gpios  
*picked_gpios = dev_get_platdata(&pdev->dev);
```

```
static struct platform_device my_device = {  
  
    .name = "my_drv_name", .id = 0,  
  
    .dev.platform_data = &my_device_pdata, .resource =  
    jz4740_udc_resources, .num_resources =  
    ARRAY_SIZE(jz4740_udc_resources), };  
  
platform_device_register(&my_device);
```

The name of the device is very important, and is used by the kernel to match the driver with the same name.

Device provisioning - the new and recommended way

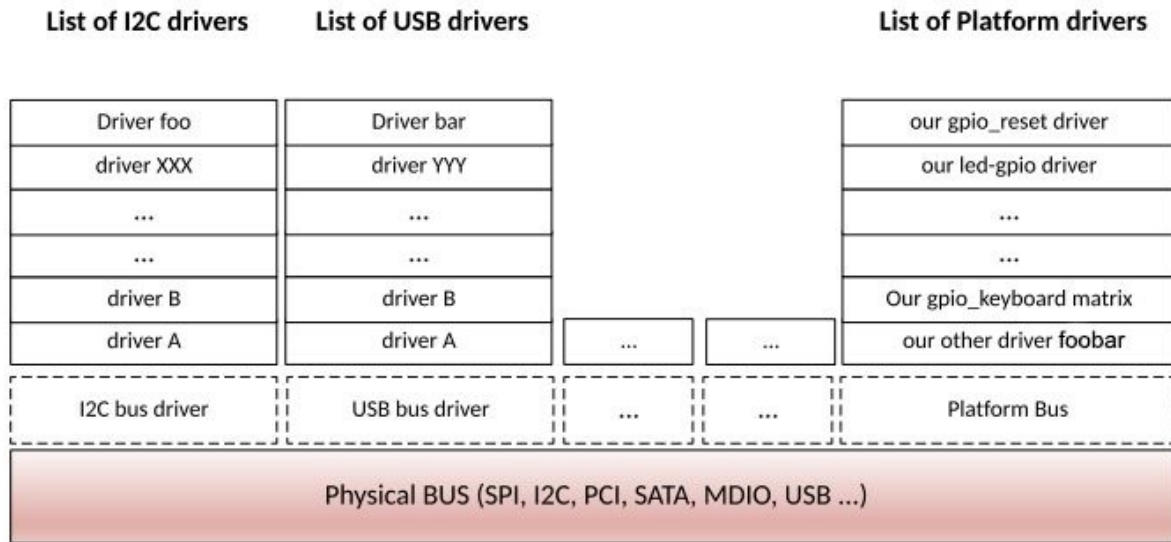
In the first method, any modification will necessitate rebuilding the whole kernel. If the kernel had to include any application/board-specific configurations, its size would incredibly increase. In order to keep things simple, and separate devices declarations (since they are not really part of the kernel) from the kernel source, a new concept has been introduced: *the device tree*. The main goal of DTS is to remove very specific and never-tested code from kernel. With the device tree, platform data and resources are homogenous. The device tree is a hardware description file and has a format similar to a tree structure, where every device is represented with a node, and any data or resource or configuration data is represented as the node's property. This way, you only need to recompile the device tree when you make some modifications. The device tree forms the subject of the next chapter, and we will see how to introduce it to the platform device.

Devices, drivers, and bus matching

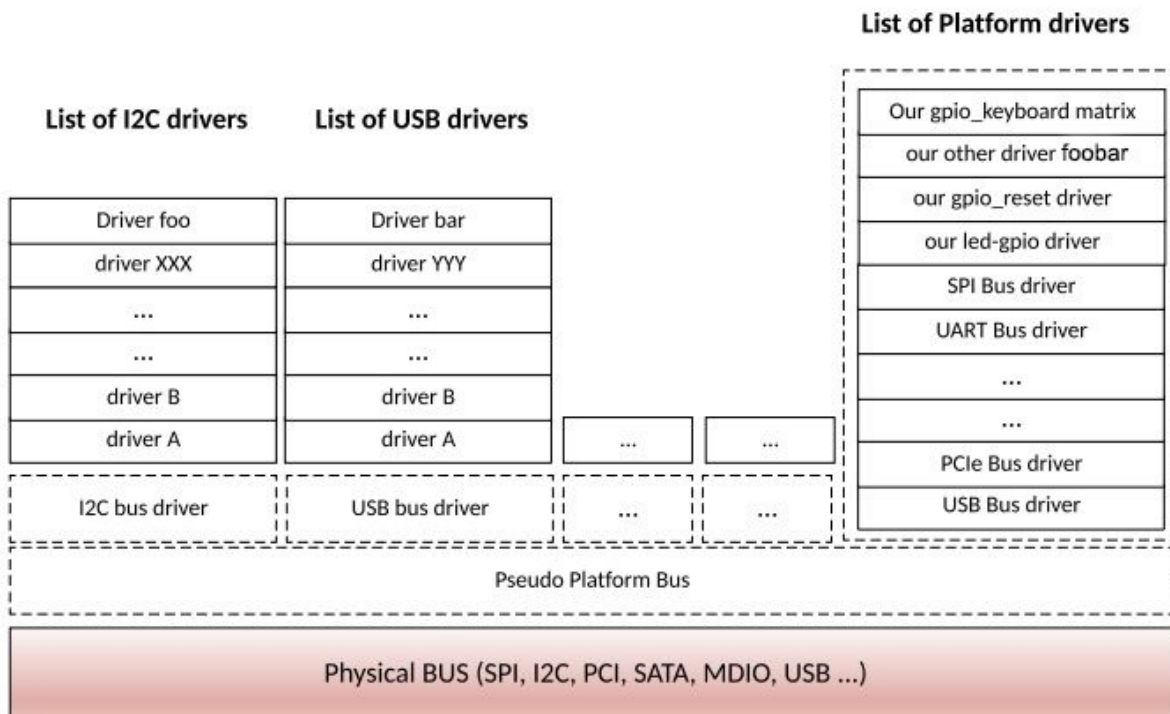
Before any match can occur, Linux calls the `platform_match(struct device *dev, struct device_driver *drv)`. Platform devices are matched with their drivers by means of strings. According to the Linux device model, the bus element is the most important part. Each bus maintains a list of drivers and devices that are registered with it. The bus driver is responsible for devices and drivers matching. Any time one connects a new device or adds a new driver to a bus, that bus starts the matching loop.

Now, suppose that you register a new I2C device using functions provided by the I2C core (discussed in next chapter). The kernel will trigger the I2C bus matching loop, by calling the I2C core match function registered with the I2C bus driver, to check if there is already a registered driver that matches with your device. If there is no match, nothing will happen. If a match occurs, the kernel will notify (by means of a communication mechanism called netlink socket) the device manager (udev/mdev), which will load (if not loaded yet) the driver your device matched with. Once the driver loads, its `probe()` function will immediately be executed. Not only does I2C work like that, but every bus has its own matching mechanism that is roughly the same. A bus matching loop is triggered at each device or driver registration.

We can sum up what we have said in the preceding section in the following figure:



Every registered drivers and devices sit on a bus. This makes a tree. USB buses may be children of PCI buses, whereas MDIO buses are generally children of other devices, and so on. Thus, our preceding figure changes as follows:



When you register a driver with the `platform_driver_probe()` function, the kernel walks through the table of registered platform devices and looks for a match. If any, it calls the matched driver's `probe` function with the platform data.

How can platform devices and platform drivers match?

So far, we have only discussed how to fill different structures of both devices and drivers. But now we will see how they are registered with the kernel, and how Linux knows which devices are handled by which driver. The answer is `MODULE_DEVICE_TABLE`. This macro lets a driver expose its ID table, which describes which devices it can support. In the meantime, if the driver can be compiled as a module, the `driver.name` field should match the module name. If it does not match, the module won't be automatically loaded, unless we have used the `MODULE_ALIAS` macro to add another name for the module. At compilation time, that information is extracted from all the drivers in order to build a device table. When the kernel has to find the driver for a device (when a matching needs to be performed), the device table is walked through by the kernel. If an entry is found matching the `compatible` (for device tree), `device/vendor id` OR `name` (for device ID table or name) of the added device, then the module providing that match is loaded (running the module's `init` function), and the `probe` function is called. The `MODULE_DEVICE_TABLE` macro is defined in `linux/module.h`:

```
| #define MODULE_DEVICE_TABLE(type, name)
```

The following is the description of each parameter given to this macro:

- `type`: This can be either `i2c`, `spi`, `acpi`, `of`, `platform`, `usb`, `pci` or any other bus which you may find in `include/linux/mod_devicetable.h`. It depends on the bus our device sits on, or on the matching mechanism we want to use.
- `name`: This is a pointer on a `xxx_device_id` array, used for device matching. If we were talking about I2C devices, the structure would be `i2c_device_id`. For SPI device, it should be `spi_device_id`, and so on. For the device tree **Open Firmware (OF)** matching mechanism, we must use `of_device_id`.



For new non-discoverable platform device drivers, it is recommended not to use platform data anymore, but to use device tree capabilities instead, with OF matching mechanism. Please do note that the two methods are not mutually exclusive, thus one can

mix these together.

Let's get deeper into the details for matching mechanisms, except for the OF style match which we will discuss in [Chapter 6](#), *The Concept of Device Tree*.

```

static int platform_match(struct device *dev, struct device_driver
*drv) {

    struct platform_device *pdev = to_platform_device(dev); struct
platform_driver *pdrv = to_platform_driver(drv);

    /* When driver_override is set, only bind to the matching driver */

    if (pdev->driver_override) return !strcmp(pdev->driver_override,
drv->name);

    /* Attempt an OF style match first */

    if (of_driver_match_device(dev, drv)) return 1;

    /* Then try ACPI style match */

    if (acpi_driver_match_device(dev, drv)) return 1;

    /* Then try to match against the id table */

    if (pdrv->id_table) return platform_match_id(pdrv->id_table, pdev)
!= NULL;

    /* fall-back to driver name match */

    return (strcmp(pdev->name, drv->name) == 0); }

static const struct platform_device_id *platform_match_id(

    const struct platform_device_id *id, struct platform_device *pdev)

```

```

{
    while (id->name[0]) {
        if (strcmp(pdev->name, id->name) == 0) {
            pdev->id_entry = id; return id;
        }
        id++;
    }

    return NULL;
}

struct device_driver {
    const char *name;

    [...]

    const struct of_device_id *of_match_table; const struct
acpi_device_id *acpi_match_table; };

```

I intentionally removed fields that we are not interested in. struct device_driver forms the basis of every device driver. Whether it is an I2C, SPI, TTY, or other device driver, they all embed a struct device_driver element.

OF style and ACPI match

OF style is explained in [Chapter 6](#), *The Concept of Device Tree*. The second mechanism is an ACPI table-based matching. We'll not discuss it at all in this book, but for your information, it uses struct `acpi_device_id`.

```

struct platform_device_id {

    char name[PLATFORM_NAME_SIZE]; kernel_ulong_t
    driver_data; };

static const struct platform_device_id imx_uart_devtype[] = {

    {

        .name = "imx1-uart", .driver_data = (kernel_ulong_t)
        &imx_uart_devdata[IMX1_UART], }, {

        .name = "imx21-uart", .driver_data = (kernel_ulong_t)
        &imx_uart_devdata[IMX21_UART], }, {

        .name = "imx6q-uart", .driver_data = (kernel_ulong_t)
        &imx_uart_devdata[IMX6Q_UART], }, {

        /* sentinel */

    }

};

static const struct platform_device_id *platform_match_id(

    const struct platform_device_id *id, struct platform_device *pdev)
{

    while (id->name[0]) {

        if (strcmp(pdev->name, id->name) == 0) {

            pdev->id_entry = id; return id;

```

```

    }

    id++;

    }

    return NULL;

}

static void serial_imx_probe_pdata(struct imx_port *sport, struct
platform_device *pdev) {

    struct imxuart_platform_data *pdata = dev_get_platdata(&pdev-
>dev);

    sport->port.line = pdev->id; sport->devdata = (struct imx_uart_data
*) pdev->id_entry->driver_data;

    if (!pdata)

    return;

    [...]

}

```

`pdev->id_entry` is a struct `platform_device_id`, which is a pointer to the matching ID table entry made available by the kernel, and whose `driver_data` element is casted back to a pointer on the data structure.

Per device-specific data on ID table matching

In the previous section, we have used

`platform_device_id.platform_data` as a pointer. Your driver may need to support more than one device type. In this situation, you will need specific device data for each device type you support. You should then use the device id as an index to an array that contains every possible device data, and not as a pointer address anymore. The following are detailed steps in an example:

1. We define an enumeration, depending on the device type that we need to support in our driver:

```
enum abx80x_chip {
    AB0801,
    AB0803,
    AB0804,
    AB0805,
    AB1801,
    AB1803,
    AB1804,
    AB1805,
    ABX80X
};
```

2. We define the specific data-type structure:

```
struct abx80x_cap {
    u16 pn;
    bool has_tc;
};
```

3. We fill an array with default values, and depending on the index in `device_id`, we can pick the right data:

```
static struct abx80x_cap abx80x_caps[] = {
    [AB0801] = {.pn = 0x0801},
    [AB0803] = {.pn = 0x0803},
    [AB0804] = {.pn = 0x0804, .has_tc = true},
    [AB0805] = {.pn = 0x0805, .has_tc = true},
    [AB1801] = {.pn = 0x1801},
    [AB1803] = {.pn = 0x1803},
```

```

    [AB1804] = {.pn = 0x1804, .has_tc = true},
    [AB1805] = {.pn = 0x1805, .has_tc = true},
    [ABX80X] = {.pn = 0}
};

```

4. We define our `platform_device_id` with a specific index:

```

static const struct i2c_device_id abx80x_id[] = {
    { "abx80x", ABX80X },
    { "ab0801", AB0801 },
    { "ab0803", AB0803 },
    { "ab0804", AB0804 },
    { "ab0805", AB0805 },
    { "ab1801", AB1801 },
    { "ab1803", AB1803 },
    { "ab1804", AB1804 },
    { "ab1805", AB1805 },
    { "rv1805", AB1805 },
    { }
};

```

5. Here we just have to do the stuff in the probe function:

```

static int rs5c372_probe(struct i2c_client *client,
const struct i2c_device_id *id)
{
    [...]

    /* We pick the index corresponding to our device */
    int index = id->driver_data;

    /*
     * And then, we can access the per device data
     * since it is stored in abx80x_caps[index]
     */
}

```

```
static struct platform_driver imx_ssi_driver = {  
  
    .probe = imx_ssi_probe, .remove = imx_ssi_remove,  
  
    /* As you can see here, only the 'name' field is filled */  
  
    .driver = {  
  
        .name = "imx-ssi", },  
  
};
```

```
module_platform_driver(imx_ssi_driver);
```

To add a device that matches this driver, one must call `platform_device_register` or `platform_add_devices`, with the same name `imx-ssi`, in the board-specific file (usually in `arch/<your_arch>/mach-*/board-*.c`). For our quad core i.MX6-based UDOO, it is `arch/arm/mach-imx/mach-imx6q.c`.

Summary

The kernel pseudo platform bus has no secrets for you anymore. With bus matching mechanisms, you are able to understand how, when, and why your driver has been loaded, as well as which device it was for. We can implement any `probe` function, based on the matching mechanism we want. Since the main purpose of a driver is to handle a device, we are now able to populate devices in the system (the old and depreciated way). To finish in beauty, the next chapter will exclusively deal with the device tree, which is the new mechanism used to populate devices, along with their configurations, on the system.

The Concept of Device Tree

The **Device Tree (DT)** is an easy to read hardware description file, with JSON like formatting style, which is a simple tree structure where devices are represented by nodes with their properties. Properties can be either empty (just the key, to describe boolean values), or key-value pairs where the value can contain an arbitrary byte stream. This chapter is a simple introduction to DT. Every kernel subsystem or framework has its own DT binding. We will talk about those specific bindings when we deal with concerned topics. The DT originated from OF, which is a standard endorsed by computer companies, and whose main purpose is defining interfaces for computer firmware systems. That said, one can find more on DT specification at <http://www.devicetree.org/>. Therefore, this chapter will cover the basics of DT, such as:

- Naming convention, as well as aliases and labeling
- Describing data types and their APIs
- Managing addressing schemes and accessing the device resources
- Implementing OF match style and providing application-specific data

```
#include <linux/of.h> #include <linux/of_device.h>

/* This is a comment */

// This is another comment node_label: nodename@reg{

    string-property = "a string"; string-list = "red fish", "blue fish"; one-
int-property = <197>; /* One cell in this property */

    int-list-property = <0xbeef 123 0xabcd4>; /*each number (cell) is a
<br/> *32 bit integer(uint32).<br/> *There are 3 cells in <br/> */this
property

    mixed-list-property = "a string", <0xadbcd45>, <35>, [0x01 0x23
0x45]

    byte-array-property = [0x01 0x23 0x45 0x67]; boolean-property; };
```

The following are some definitions of data types used in device trees:

- Text strings are represented with double quotes. One can use commas to create a list of the strings.
- Cells are 32-bit unsigned integers delimited by angle brackets.
- Boolean data is nothing but an empty property. The true or false value depends on the property being there or not.

```
expander@20 {  
    compatible = "microchip,mcp23017"; reg = <20>; [...]  
};  
  
i2c@021a0000 {  
    compatible = "fsl,imx6q-i2c", "fsl,imx21-i2c"; reg = <0x021a0000  
0x4000>; [...]  
};
```

On the other hand, the `label` is optional. Labeling a node is useful only if the node is intended to be referenced from a property of another node. One can see a label as a pointer to node, as explained in the next section.

Aliases, labels, and phandle

It is very important to understand how these three elements work. They are frequently used in the DT. Let us take the following DT to explain how they work:

```
aliases {
    ethernet0 = &fec;
    gpio0 = &gpio1;
    gpio1 = &gpio2;
    mmc0 = &usdhc1;
    [...]
};
gpio1: gpio@0209c000 {
    compatible = "fsl,imx6q-gpio", "fsl,imx35-gpio";
    [...]
};
node_label: nodename@reg {
    [...];
    gpios = <&gpio1 7 GPIO_ACTIVE_HIGH>;
};
```

A label is nothing but a way to tag a node, to let the node be identified by a unique name. In the real world, that name is converted into a unique 32-bit value by the DT compiler. In the preceding example, `gpio1` and `node_label` are both labels. Labels can then be used to refer to a node, since a label is unique to a node.

A **pointer handle (phandle)** is a 32-bit value associated with a node that is used to uniquely identify that node so that the node can be referenced from a property in another node. Labels are used to have a pointer to the node. By using `<&mylabel>`, you point to the node whose label is `mylabel`.



The use of & is just like in the C programming language; to obtain the address of an element.

In the preceding example, `&gpio1` is converted to the phandle so that it refers to `gpio1` node. The same goes for the following example:

```
thename@address {
    property = <&mylabel>;
};
```



```
| mylabel: thename@adresses {  
|     [...]   
| }
```

In order not to walk through the whole tree to look for a node, the concept of aliases has been introduced. In the DT, the `aliases` node can be seen like a quick lookup table, an index of another node. One can use the function `find_node_by_alias()` to find a node given its alias. The aliases are not used directly in the DT source, but are instead deferred by the Linux kernel.

DT compiler

The DT comes in two forms: the textual form, which represents the sources also known as `DTs`, and the binary blob form, which represents the compiled DT, also known as `DTB`. Source files have the `.dts` extension. Actually, there are also `.dtsi` text files, which represent SoC level definitions, whereas `.dts` files represent board level definitions. One can see `.dtsi` as header files, that should be included in `.dts` one, which are source files, not the reverse, a bit like including header files (`.h`) in the source file (`.c`). On the other hand, binary files use the `.dtb` extension.

There is actually a third form, which is the runtime representation of the DT in `/proc/device-tree`.

As its name says, the tool used to compile the device tree is called the **device tree compiler (dtt)**. From the root kernel source, one can compile either a standalone specific DT or all DTs for the specific architecture.

Let us compile all DT (`.dts`) files for arm SoC's:

```
| ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make dtbs
```

For a standalone DT:

```
| ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make imx6dl-sabrelite.dtb
```

In the preceding example, the name of the source file is `imx6dl-sabrelite.dts`.

Given a compiled device tree (`.dtb`) file, you can do the reverse operation and extract the source (`.dts`) file: `dtt -I dtb -O dtsarch/arm/boot/dts imx6dl-sabrelite.dtb >path/to/my_devicetree.dts`



For the purpose of debugging, it could be useful to expose the DT to the user space. The `CONFIG_PROC_DEVICETREE` configuration variable will do that for you. You can then explore and walk through the DT in `/proc/device-tree`.

Representing and addressing devices

Each device is given at least one node in the DT. Some properties are common to many device types, especially devices sitting on a bus known to the kernel (SPI, I2C, platform, MDIO, and so on). These properties are `reg`, `#address-cells`, and `#size-cells`. The purpose of these properties is device addressing on the bus they sit on. That said, the main addressing property is `reg`, which is a generic property and whose meaning depends on the bus the device sits on. The # (sharp) that prefixes `size-cell` and `address-cell` can be translated into `length`.

Each addressable device gets a `reg` property that is a list of tuples in the form `reg = <address0size0 [address1size1] [address2size2] ... >`, where each tuple represents an address range used by the device. `#size-cells` indicates how many 32 bit cells are used to represent size, and may be 0 if size is not relevant. On the other hand, `#address-cells` indicates how many 32 bit cells are used to represent address. In other word, the address element of each tuple is interpreted according to `#address-cell`; same for the size element, which is interpreted according to `#size-cell`.

Actually, addressable devices inherit from `#size-cell` and `#address-cell` of their parent, which is the node that represents the bus controller. The presence of `#size-cell` and `#address-cell` in a given device does not affect the device itself, but its children. In other words, before interpreting the `reg` property of a given node, one must know the parent node's `#address-cells` and `#size-cells` values. The parent node is free to define whatever addressing scheme is suitable for device sub-nodes (children).

&i2c3 {

[...]

status = "okay";

temperature-sensor@49 {

compatible = "national,lm73"; reg = <0x49>; };

pcf8523: rtc@68 {

compatible = "nxp,pcf8523"; reg = <0x68>; };

};

&ecspi1 {

fsl,spi-num-chipselects = <3>; cs-gpios = <&gpio5 17 0>, <&gpio5 17 0>, <&gpio5 17 0>; status = "okay";

[...]

ad7606r8_0: ad7606r8@1 {

compatible = "ad7606-8"; reg = <1>; spi-max-frequency = <1000000>; interrupt-parent = <&gpio4>; interrupts = <30 0x0>; convst-gpio = <&gpio6 18 0>; };

};

If one looks at a SoC level file at `arch/arm/boot/dts/imx6qdl.dtsi`, one will notice that `#size-cells` and `#address-cells` are respectively set to 0 for the former, and 1 for the last, in both `i2c` and `spi` nodes, which are respectively parents of I2C and SPI devices enumerated in the preceding section. This helps us to understand their `reg` property, which is only one cell for the address value, and none for the size value.

I2C device's `reg` property is used to specify the device's address on the bus. For SPI devices, `reg` represents the index of the chip-select line assigned to the device among the list of chips-select the controller node has. For example, for the `ad7606r8` ADC, the chip-select index is 1, which corresponds to `<gpio5 17 0>` in `cs-gpios`, which is the list of chip-select of the controller node.

You may ask why I used the I2C/SPI node's phandle: the answer is because I2C/SPI devices should be declared at board level file (`.dts`), whereas I2C/SPI buses controller are declared at SoC level file (`.dtsi`).

```
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>; [...];
```

```
        spba-bus@02000000 {
```

```
            compatible = "fsl,spba-bus", "simple-bus"; #address-cells = <1>;  
            #size-cells = <1>; reg = <0x02000000 0x40000>; [...]
```

```
        ecspi1: ecspi@02008000 {
```

```
            #address-cells = <1>; #size-cells = <0>; compatible = "fsl,imx6q-  
            ecspi", "fsl,imx51-ecspi"; reg = <0x02008000 0x4000>; [...]
```

```
        };
```

```
        i2c1: i2c@021a0000 {
```

```
            #address-cells = <1>; #size-cells = <0>; compatible = "fsl,imx6q-  
            i2c", "fsl,imx21-i2c"; reg = <0x021a0000 0x4000>; [...]
```

```
        };
```

```
    };
```

};

In the preceding example, child nodes whose parent has `simple-bus` in the `compatible` property will be registered as platform devices. One can also see how I2C and SPI bus controllers change the addressing scheme of their children by setting `#size-cells = <0>`; because it is not relevant to them. A well-known place to look for any binding information is in the kernel device tree's documentation:

[Documentation/devicetree/bindings/](#)

```
uart1: serial@02020000 {  
  
    compatible = "fsl,imx6q-uart", "fsl,imx21-uart"; reg =  
    <0x02020000 0x4000>; interrupts = <0 26  
    IRQ_TYPE_LEVEL_HIGH>; clocks = <&clks  
    IMX6QDL_CLK_UART_IPG>, <&clks  
    IMX6QDL_CLK_UART_SERIAL>; clock-names = "ipg", "per";  
    dmas = <&sdma 25 4 0>, <&sdma 26 4 0>; dma-names = "rx", "tx";  
  
    status = "disabled"; };
```



```
fake_device {
    compatible = "packt,fake-device"; reg = <0x4a064000 0x800>,
    <0x4a064800 0x200>, <0x4a064c00 0x200>; reg-names = "config",
    "ohci", "ehci"; interrupts = <0 66 IRQ_TYPE_LEVEL_HIGH>, <0
    67 IRQ_TYPE_LEVEL_HIGH>; interrupt-names = "ohci", "ehci";
    clocks = <&clks IMX6QDL_CLK_UART_IPG>, <&clks
    IMX6QDL_CLK_UART_SERIAL>; clock-names = "ipg", "per";
    dmas = <&sdma 25 4 0>, <&sdma 26 4 0>; dma-names = "rx", "tx";
};
```

```
struct resource *res1, *res2;
```

```
res1 = platform_get_resource_byname(pdev, IORESOURCE_MEM,
    "ohci"); res2 = platform_get_resource_byname(pdev,
    IORESOURCE_MEM, "config");
```

```
struct dma_chan *dma_chan_rx, *dma_chan_tx;
```

```
dma_chan_rx = dma_request_slave_channel(&pdev->dev, "rx");
dma_chan_tx = dma_request_slave_channel(&pdev->dev, "tx");
```

```
int txirq, rxirq;
```

```
txirq = platform_get_irq_byname(pdev, "ohci"); rxirq =
platform_get_irq_byname(pdev, "ehci");
```

```
struct clk *clk_per, *clk_ipg;
```

```
clk_ipg = devm_clk_get(&pdev->dev, "ipg"); clk_per =
devm_clk_get(&pdev->dev, "pre");
```

This way, you are sure to map the right name to the right resource, without needing to play with the index anymore.

```

struct resource *res; void __iomem *base;

res = platform_get_resource(pdev, IORESOURCE_MEM, 0); /*
 * Here one can request and map the memory region * using
 request_mem_region(res->start, resource_size(res), pdev->name) *
 and ioremap(iores->start, resource_size(iores) *

 * These function are discussed in chapter 11, Kernel Memory
 Management.

 */

base = devm_ioremap_resource(&pdev->dev, res); if (IS_ERR(base))

    return PTR_ERR(base);

```

platform_get_resource() will set the start and end fields of struct res according to the memory region present in the first (index 0) reg assignment. Please remember the last argument of platform_get_resource() represents the resource index. In the preceding sample, 0 indexes the first value of that resource type, just in case the device is assigned more than one memory region in the DT node. In our example, it's reg = <0x02020000 0x4000>, meaning that the allocated region starts at physical address 0x02020000 and has the size of 0x4000 bytes. platform_get_resource() will then set res.start = 0x02020000 and res.end = 0x02023fff.

Handling interrupts

The interrupt interface is actually divided into two parts; the consumer side and the controller side. Four properties are used to describe interrupt connections in the DT: The controller is the device that exposes IRQ lines to the consumer. In controller side, on has the following properties:

- `interrupt-controller`: An empty (Boolean) property that one should define 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. Consumer binding expects the following properties:

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

Interrupt binding and interrupt specifiers are tied to the interrupt controller device. The number of cells used to define an interrupt input depends on the interrupt controller, which is the only one deciding, by mean of its `#interrupt-cells` property. In the case of i.MX6, the interrupt controller is a **Global Interrupt Controller (GIC)**. Its binding is well explained in *Documentation/devicetree/bindings/arm/gic.txt*.

The interrupt handler

This consist of fetching the IRQ number from the DT, and mapping it into Linux IRQ, thus registering a function callback for it. The driver code to do this is quite simple: `int irq = platform_get_irq(pdev, 0); ret = request_irq(irq, imx_rxint, 0, dev_name(&pdev->dev), sport);`

The `platform_get_irq()` call will return the `irq` number; this number is usable by `devm_request_irq()` (`irq` is then visible in `/proc/interrupts`). The second argument, `0`, says that we need the first interrupt specified in the device node. If, there is more than one interrupt, we can change this index according to the interrupt we need, or just use the named resource.

In our preceding example, the device node contains an interrupt specifier, which looks like as follows: `interrupts = <0 66 IRQ_TYPE_LEVEL_HIGH>;`

- According to ARM GIC, the first cell informs us about interrupt type:
 - **0: Shared peripheral interrupt (SPI)**, for interrupts signal shared among cores, which can be routed by the GIC to any core
 - **1: Private peripheral interrupt (PPI)**, for interrupt signal private to an individual core

The documentation can be found at: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0407e/CCHDBEBE.html>.

- The second cell holds the interrupt number. This number depends on whether the interrupt line is a PPI or SPI.
- The third cell, `IRQ_TYPE_LEVEL_HIGH` in our case, represents sense level. All of the available sense levels are defined in `include/linux/irq.h`.

Interrupt controller code

The `interrupt-controller` property is used to declare a device as an interrupt controller. The `#interrupt-cells` property defines how many cells must be used to define a single interrupt line. We will discuss this in detail in [Chapter 16](#), *Advanced IRQ Management* .

```
node_label: nodename@reg{
```

```
    string-property = ""a string""; string-list = ""red fish"", ""blue  
fish""; one-int-property = <197>; /* One cell in this property */
```

```
    int-list-property = <0xbeef 123 0xabcd4>;/* each number (cell) is  
32 a * bit integer(uint32). There * are 3 cells in this property */
```

```
    mixed-list-property = "a string", <0xadbcd45>, <35>, [0x01 0x23  
0x45]
```

```
    byte-array-property = [0x01 0x23 0x45 0x67]; one-cell-property =  
<197>; boolean-property;
```

```
};
```

Text string

The following is one `string` property: `string-property = "a string";`

Back in the driver, one should use `of_property_read_string()` to read a string value. Its prototype is defined as follows: `int of_property_read_string(const struct device_node *np, const char *propname, const char **out_string)`

The following code shows how you can use it:

```
| const char *my_string = NULL;  
| of_property_read_string(pdev->dev.of_node, "string-property", &my_string);
```

Cells and unsigned 32-bit integers

The following are our `int` properties: `one-int-property = <197>; int-list-property = <1350000 0x54dae47 1250000 1200000>;`

One should use `of_property_read_u32()` to read a cell value. Its prototype is defined as follows: `int of_property_read_u32_index(const struct device_node *np, const char *propname, u32 index, u32 *out_value)`

Back in the driver,

```
    unsigned int number;  
    of_property_read_u32(pdev->dev.of_node, "one-cell-property", &number);
```

One can use `of_property_read_u32_array` to read a list of cells. Its prototype is as follows: `int of_property_read_u32_array(const struct device_node *np, const char *propname, u32 *out_values, size_t sz);`

Here, `sz` is the number of array elements to read. Have a look at `drivers/of/base.c` to see how to interpret its return value: `unsigned int cells_array[4]; if (of_property_read_u32_array(pdev->dev.of_node, "int-list-property", cells_array, 4)) { dev_err(&pdev->dev, "list of cells not specified\n"); return -EINVAL; }`

Boolean

One should use `of_property_read_bool()` to read the Boolean property whose name is given in the second argument of the function:

```
bool my_bool = of_property_read_bool(pdev->dev.of_node, "boolean-property");
if(my_bool){
    /* boolean is true */
} else
    /* Boolean is false */
}
```

```

eeeprom: ee24lc512@55 {

    compatible = "microchip,24xx512"; reg = <0x55>;

    partition1 {

        read-only; part-name = "private"; offset = <0>; size = <1024>; };

    partition2 {

        part-name = "data"; offset = <1024>; size = <64512>; };

    };

    struct device_node *np = pdev->dev.of_node; struct device_node
    *sub_np; for_each_child_of_node(np, sub_np) {

        /* sub_np will point successively to each sub-node */

        [...]

        int size;

        of_property_read_u32(client->dev.of_node, "size", &size);

        ...

    }

```

Platform drivers and DT

Platform drivers also work with DT. That being said, it is the recommended way to deal with platform devices nowadays, and there is no need to touch board files anymore, or even to recompile the kernel when a device's property changes. If you remember, in the previous chapter we discussed OF match style, which is a matching mechanism based on the DT. Let us see in the following section how it works:

OF match style

OF match style is the first matching mechanism performed by the platform core in order to match devices with their drivers. It 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. Any platform driver that declares one of the strings listed in the `compatible` property will trigger a match and will see its `probe` function executed.

A DT match entry is described in the kernel as an instance of the `struct of_device_id` structure, which is defined in `linux/mod_devicetable.h` and looks like:

```
// we are only interested in the two last elements of the structure
struct of_device_id {
    [...]
    char compatible[128];
    const void *data;
};
```

The following is the meaning of each element of the structure:

- `char compatible[128]`: This is the string used to match the device node's `compatible` property in the DT. They must be identical before a match occurs.
- `const void *data`: This can point to any structure, which can be used as per-device type configuration data.

Since the `of_match_table` is a pointer, you can pass an array of the `struct of_device_id` to make your driver compatible with more than one device:

```
static const struct of_device_id imx_uart_dt_ids[] = {
    { .compatible = "fsl,imx6q-uart", },
    { .compatible = "fsl,imx1-uart", },
    { .compatible = "fsl,imx21-uart", },
    { /* sentinel */ }
};
```

Once you have filled your array of ids, it must be passed to the `of_match_table` field of your platform driver, in the driver substructure:

```
| static struct platform_driver serial_imx_driver = {
```

```

    [...]
    .driver      = {
        .name     = "imx-uart",
        .of_match_table = imx_uart_dt_ids,
        [...]
    },
};

```

At this step, only your driver is aware of your `of_device_id` array. To get the kernel informed too (so that it can store your IDs in the device list maintained by the platform core), your array has to be registered with `MODULE_DEVICE_TABLE`, as described in [Chapter 5, Platform Device Drivers](#):

```

| MODULE_DEVICE_TABLE(of, imx_uart_dt_ids);

```

That is all! Our driver is DT-compatible. Back in our DT, let's declare a device compatible with our driver:

```

| uart1: serial@02020000 {
|     compatible = "fsl,imx6q-uart", "fsl,imx21-uart";
|     reg = <0x02020000 0x4000>;
|     interrupts = <0 26 IRQ_TYPE_LEVEL_HIGH>;
|     [...]
| };

```

Two compatible strings are provided here. If the first one does not match any driver, the core will perform the match with the second.

When a match occurs, the `probe` function of your driver is called, with a `struct platform_device` structure as the parameter, which contains a `struct device dev` field, in which there is a field `struct device_node *of_node` that corresponds to the node associated to our device, so that one can use it to extract the device settings:

```

| static int serial_imx_probe(struct platform_device *pdev)
| {
|     [...]
|     struct device_node *np;
|     np = pdev->dev.of_node;
|
|     if (of_get_property(np, "fsl,dte-mode", NULL))
|         sport->dte_mode = 1;
|     [...]
| }

```

One can check if the DT node is set to know whether the driver has been loaded in response to an `of_match`, or instantiated from within the board's `init` file. You should then use the `of_match_device` function, in order to pick the `struct *of_device_id` entry that originated the match, which may contain the specific data you have

passed:

```
static int my_probe(struct platform_device *pdev)
{
    struct device_node *np = pdev->dev.of_node;
    const struct of_device_id *match;

    match = of_match_device(imx_uart_dt_ids, &pdev->dev);
    if (match) {
        /* Devicetree, extract the data */
        my_data = match->data
    } else {
        /* Board init file */
        my_data = dev_get_platdata(&pdev->dev);
    }
    [...]
}
```

```
#ifdef CONFIG_OF
```

```
static const struct of_device_id imx_uart_dt_ids[] = {
```

```
    { .compatible = "fsl,imx6q-uart", }, { .compatible = "fsl,imx1-uart",  
}, { .compatible = "fsl,imx21-uart", }, { /* sentinel */ }
```

```
};
```

```
/* other devicetree dependent code */
```

```
[...]
```

```
#endif
```

```
#define of_match_ptr(_ptr) (_ptr) /* When CONFIG_OF is enabled */
```

```
#define of_match_ptr(_ptr) NULL /* When it is not */
```

```
static int my_probe(struct platform_device *pdev) {
```

```
    const struct of_device_id *match; match =  
of_match_device(of_match_ptr(imx_uart_dt_ids), &pdev->dev); [...]
```

```
}
```

```
static struct platform_driver serial_imx_driver = {
```

```
[...]
```

```
.driver = {
```

```
.name = "imx-uart", .of_match_table =
```

```
of_match_ptr(imx_uart_dt_ids), },  
  
};
```

This eliminates having a `#ifdef`, returning `NULL` when `OF` is disabled.

Support multiple hardware with per device-specific data

Sometimes, a driver can support different hardware, each with its specific configuration data. That data may be dedicated function tables, specific register values, or anything unique to each hardware. The following example describes a generic approach:

Let us first remember what `struct of_device_id` looks like, in

`include/linux/mod_devicetable.h`.

```
/*
 * Struct used for matching a device
 */
struct of_device_id {
    [...]
    char    compatible[128];
    const void *data;
};
```

The field we are interested in is `const void *data`, so we can use it to pass any data for each specific device.

Let's say we own three different devices, each with a specific private data. `of_device_id.data` will contain a pointer to specific parameters. This example is inspired by `drivers/tty/serial/imx.c`.

First, we declare private structures:

```
/* i.MX21 type uart runs on all i.mx except i.MX1 and i.MX6q */
enum imx_uart_type {
    IMX1_UART,
    IMX21_UART,
    IMX6Q_UART,
};

/* device type dependent stuff */
struct imx_uart_data {
    unsigned uts_reg;
    enum imx_uart_type devtype;
};
```

Then we fill an array with each device-specific data:

```
static struct imx_uart_data imx_uart_devdata[] = {
    [IMX1_UART] = {
        .uts_reg = IMX1_UTS,
        .devtype = IMX1_UART,
    },
    [IMX21_UART] = {
        .uts_reg = IMX21_UTS,
        .devtype = IMX21_UART,
    },
    [IMX6Q_UART] = {
        .uts_reg = IMX21_UTS,
        .devtype = IMX6Q_UART,
    },
};
```

Each compatible entry is tied with a specific array index:

```
static const struct of_device_id imx_uart_dt_ids[] = {
    { .compatible = "fsl,imx6q-uart", .data = &imx_uart_devdata[IMX6Q_UART], },
    { .compatible = "fsl,imx1-uart", .data = &imx_uart_devdata[IMX1_UART], },
    { .compatible = "fsl,imx21-uart", .data = &imx_uart_devdata[IMX21_UART], },
    { /* sentinel */ }
};
MODULE_DEVICE_TABLE(of, imx_uart_dt_ids);

static struct platform_driver serial_imx_driver = {
    [...]
    .driver = {
        .name = "imx-uart",
        .of_match_table = of_match_ptr(imx_uart_dt_ids),
    },
};
```

Now in the `probe` function, whatever the match entry is, it will hold a pointer to the device-specific structure:

```
static int imx_probe_dt(struct platform_device *pdev)
{
    struct device_node *np = pdev->dev.of_node;
    const struct of_device_id *of_id =
        of_match_device(of_match_ptr(imx_uart_dt_ids), &pdev->dev);

    if (!of_id)
        /* no device tree device */
        return 1;
    [...]
    sport->devdata = of_id->data; /* Get private data back */
}
```

In the preceding code, `devdata` is an element of a structure in the original source, and declared like `const struct imx_uart_data *devdata;` we could have stored any specific parameter in the array.

```

static const struct platform_device_id sdma_devtypes[] = {

    {

        .name = "imx51-sdma", .driver_data = (unsigned
long)&sdma_imx51, }, {

        .name = "imx53-sdma", .driver_data = (unsigned
long)&sdma_imx53, }, {

        .name = "imx6q-sdma", .driver_data = (unsigned
long)&sdma_imx6q, }, {

        .name = "imx7d-sdma", .driver_data = (unsigned
long)&sdma_imx7d, }, {

        /* sentinel */

    }

};

MODULE_DEVICE_TABLE(platform, sdma_devtypes);

static const struct of_device_id sdma_dt_ids[] = {

    { .compatible = "fsl,imx6q-sdma", .data = &sdma_imx6q, }, {
.compatible = "fsl,imx53-sdma", .data = &sdma_imx53, }, {
.compatible = "fsl,imx51-sdma", .data = &sdma_imx51, }, {
.compatible = "fsl,imx7d-sdma", .data = &sdma_imx7d, }, { /*
sentinel */ }

};

```

```

MODULE_DEVICE_TABLE(of, sdma_dt_ids);

static int sdma_probe(struct platform_device *pdev) {

const struct of_device_id *of_id =

of_match_device(of_match_ptr(sdma_dt_ids), &pdev->dev);
struct device_node *np = pdev->dev.of_node;

    /* If devicetree, */

    if (of_id)

drvdata = of_id->data;

    /* else, hard-coded */

    else if (pdev->id_entry) drvdata = (void *)pdev->id_entry-
>driver_data;

    if (!drvdata) {

dev_err(&pdev->dev, "unable to find driver data\n"); return -
EINVAL; }

    [...]

}

static struct platform_driver sdma_driver = {

    .driver = {

        .name = "imx-sdma", .of_match_table =
of_match_ptr(sdma_dt_ids), },

```

```
.id_table = sdma_devtypes, .remove = sdma_remove, .probe =  
sdma_probe, };
```

```
module_platform_driver(sdma_driver);
```

```

uart1: serial@02020000 {

    compatible = "fsl,imx6q-uart", "fsl,imx21-uart"; reg =
<0x02020000 0x4000>;

    interrupts = <0 26 IRQ_TYPE_LEVEL_HIGH>; dmas = <&sdma
25 4 0>, <&sdma 26 4 0>; dma-names = "rx", "tx";

};

static int my_probe(struct platform_device *pdev) {

struct iio_dev *indio_dev;

struct resource *mem, *dma_res;

struct xadc *xadc;

int irq, ret, dmareq;


    /* irq */

    irq = platform_get_irq(pdev, 0);

    if (irq <= 0) return -ENXIO; [...]


    /* memory region */

    mem = platform_get_resource(pdev, IORESOURCE_MEM, 0); xadc-
>base = devm_ioremap_resource(&pdev->dev, mem); /*

```

```
    * We could have used * devm_ioremap(&pdev->dev, mem->start,  
resource_size(mem)); * too.
```

```
    */
```

```
    if (IS_ERR(xadc->base)) return PTR_ERR(xadc->base); [...]
```

```
    /* second dma channel */
```

```
    dma_res = platform_get_resource(pdev, IORESOURCE_DMA, 1);  
    dmareq = dma_res->start;
```

```
    [...]
```

```
}
```

```
int platform_get_irq(struct platform_device *dev, unsigned int num) {
```

```
    [...]
```

```
    struct resource *r; if (IS_ENABLED(CONFIG_OF_IRQ) &&dev->  
>dev.of_node) {
```

```
        int ret;
```

```
        ret = of_irq_get(dev->dev.of_node, num); if (ret > 0 || ret == -  
EPROBE_DEFER) return ret; }
```

```
    r = platform_get_resource(dev, IORESOURCE_IRQ, num); if (r
&& r->flags & IORESOURCE_BITS) {
```

```
    struct irq_data *irqd; irqd = irq_get_irq_data(r->start); if (!irqd)
return -ENXIO; irqd_set_trigger_type(irqd, r->flags &
IORESOURCE_BITS); }
```

```
    return r ? r->start : -ENXIO; }
```

```
static struct platform_device *of_platform_device_create_pdata(
```

```
    struct device_node *np, const char *bus_id, void *platform_data,
struct device *parent)
```



```

#define SIRFSOC_I2C0MOD_PA_BASE 0xcc0e0000

#define SIRFSOC_I2C0MOD_SIZE 0x10000

#define IRQ_I2C0

static struct resource sirfsoc_i2c0_resource[] = {

    {

        .start = SIRFSOC_I2C0MOD_PA_BASE, .end =
SIRFSOC_I2C0MOD_PA_BASE + SIRFSOC_I2C0MOD_SIZE - 1,
        .flags = IORESOURCE_MEM, },{

        .start = IRQ_I2C0, .end = IRQ_I2C0, .flags = IORESOURCE_IRQ,
        }, };

i2c0: i2c@cc0e0000 {

    compatible = "sirf,marco-i2c"; reg = <0xcc0e0000 0x10000>;
    interrupt-parent = <&phandle_to_interrupt_controller_node>
    interrupts = <0 24 0>; #address-cells = <1>; #size-cells = <0>; status
    = "disabled"; };

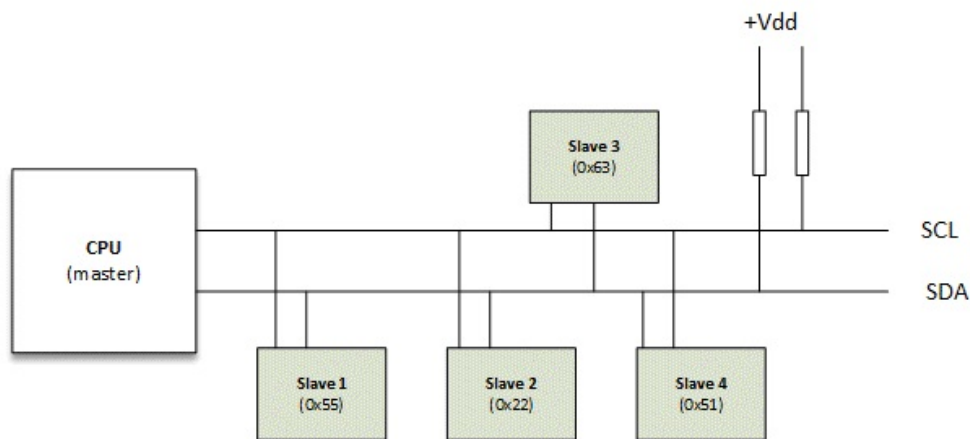
```

Summary

The time to switch from hardcoded device configuration to DT has come. This chapter gave you all you need to handle DTs. Now you have the necessary skills to customize or add whatever node and property you want into the DT, and extract them from within your driver. In the next chapter, we will talk about the I2C driver, and use the DT API to enumerate and configure our I2C devices.

I2C Client Drivers

I2C bus, invented by Philips (now NXP) is a two-wire: **Serial Data (SDA)**, **Serial Clock (SCL)** asynchronous serial bus. It is a multi-master bus, though multi-master 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 none of these can drive its output high without having pull-up resistors. SCL is generated by the master in order to synchronize data (carried by SDA) transfer over the bus. Both slave and master can send data (not at the same time of course), thus making SDA a bidirectional line. That said the SCL signal is also bidirectional, since slave can stretch the clock by keeping the SCL line low. The bus is controlled by the master, which in our case is a part of the SoC. This bus is frequently used in embedded systems to connect serial EEPROM, RTC chips, GPIO expander, temperature sensors, and so on:



I2C bus and devices

Clock speed varies from 10 KHz to 100 KHz, and 400 KHz to 2 MHz. We will not cover bus specifications or bus drivers in this book. However, it is up to the bus driver to manage the bus and take care of the specifications. An example of a bus driver for the i.MX6 chip can be found at `drivers/i2c/busses/i2c-imx.c` in kernel source, and I2C specifications can be found at http://www.nxp.com/documents/user_manual/UM10204.pdf.

In this chapter, we are interested in client drivers, in order to handle slave devices seated on the bus. The chapter will cover the following topics:

- I2C client driver architecture
- Accessing the device, thus reading/writing data from/to device
- Declaring clients from DT

The driver architecture

When the device for which you write the driver takes a seat on a physical bus called the *bus controller*, it must rely on the driver of that bus called the *controller driver*, responsible for sharing bus access between devices. The controller driver offers an abstraction layer between your device and the bus. Whenever you perform a transaction (read or write) on an I2C or USB bus for example, the I2C/USB bus controller transparently takes care of that in the background. Every bus controller driver exports a set of functions to ease the development of drivers for devices sitting on that bus. This works for every physical bus (I2C, SPI, USB, PCI, SDIO, and so on).

An I2C driver is represented in the kernel as an instance of `struct i2c_driver`. The I2C client (which represents the device itself) is represented by a `struct i2c_client` structure.

```

struct i2c_driver {

    /* Standard driver model interfaces */

    int (*probe)(struct i2c_client *, const struct i2c_device_id *); int
    (*remove)(struct i2c_client *);

    /* driver model interfaces that don't relate to enumeration */

    void (*shutdown)(struct i2c_client *);

    struct device_driver driver; const struct i2c_device_id *id_table; };

```

The struct `i2c_driver` structure contains and characterizes general access routines, needed to handle the devices claiming the driver, whereas struct `i2c_client` contains device-specific information, like its address. A struct `i2c_client` structure represents and characterizes an I2C device. Later in this chapter, we will see how to populate these structures.

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

As you can see, its parameters are:

- `struct i2c_client` pointer: This represents the I2C device itself. This structure inherits from the structure device, and is provided to your probe function by the kernel. The client structure is defined in `include/linux/i2c.h`. Its definition is as follows:

```
struct i2c_client {
    unsigned short flags; /* div., see below */
    unsigned short addr; /* chip address - NOTE: 7bit
                          /* addresses are stored in the
                          /* _LOWER_ 7 bits
    char name[I2C_NAME_SIZE];
    struct i2c_adapter *adapter; /* the adapter we sit on
    struct device dev; /* the device structure
    int irq; /* irq issued by device
    struct list_head detected;
    #if IS_ENABLED(CONFIG_I2C_SLAVE)
        i2c_slave_cb_t slave_cb; /* callback for slave mode
    #endif
};
```

- All fields are filled by the kernel, based on the parameter you provided to register the client. We will see later how to register a device to the kernel.
- `struct i2c_device_id` pointer: This points to the I2C device ID entry that matched the device that is being probed.

```

/* set the data */

void i2c_set_clientdata(struct i2c_client *client, void *data);

/* get the data */

void *i2c_get_clientdata(const struct i2c_client *client);

/* This is the device specific data structure */

struct mc9s08dz60 {

    struct i2c_client *client; struct gpio_chip chip; };

static int mc9s08dz60_probe(struct i2c_client *client, const struct
i2c_device_id *id)

{

    struct mc9s08dz60 *mc9s; if (!i2c_check_functionality(client-
>adapter, I2C_FUNC_SMBUS_BYTE_DATA)) return -EIO; mc9s =
devm_kzalloc(&client->dev, sizeof(*mc9s), GFP_KERNEL); if
(!mc9s) return -ENOMEM;

    [...]

    mc9s->client = client; i2c_set_clientdata(client, mc9s);

    return gpiochip_add(&mc9s->chip); }

```

Actually, these functions are not really specific to I2C. They do nothing but get/set the void *driver_data pointer that is a member of

the struct `device`, and itself is a member of struct `i2c_client`. In fact, we could have used `dev_get_drvdata` and `dev_set_drvdata` directly. One can see their definitions in `linux/include/linux/i2c.h`.

```

static int foo_remove(struct i2c_client *client)

static int mc9s08dz60_remove(struct i2c_client *client) {

    struct mc9s08dz60 *mc9s;

    /* We retrieve our private data */

    mc9s = i2c_get_clientdata(client);

    /* Wich hold gpiochip we want to work on */

    return gpiochip_remove(&mc9s->chip); }

```

The remove function has the responsibility to unregister us from the subsystem where we have registered in the probe() function. In the preceding example, we simply remove the gpiochip from the kernel.

```
module_i2c_driver(foo_driver);
```

```

struct i2c_device_id {

    char name[I2C_NAME_SIZE]; kernel_ulong_t driver_data; /* Data
private to the driver */

};

static struct i2c_device_id foo_idtable[] = {

    { "foo", my_id_for_foo }, { "bar", my_id_for_bar }, { }

};

MODULE_DEVICE_TABLE(i2c, foo_idtable);

static struct i2c_driver foo_driver = {

    .driver = {

        .name = "foo", },

    .id_table = foo_idtable, .probe = foo_probe, .remove = foo_remove,
}

```

Accessing the client

Serial bus transactions are just a matter of accessing registers to set/get their content. I2C respects that principle. I2C core provides two kind of API, one for plain I2C communications, and another for SMBUS compatible device, which also works with I2C devices, but not the reverse.

Plain I2C communication

The following are essential functions one usually deal with when talking to I2C devices:

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

Almost all I2C communication functions take a `struct i2c_client` as the first parameter. The second parameter contains the bytes to read or write and the third represents the number of bytes to read or write. Like any read/write function, the returned value is the number of bytes being read/written. One can also process message transfers with:

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

`i2c_transfer` sends a set of messages where each can be either a read or a write operation, and can be mixed in any way. Remember that there is no stop bit between each transaction. Looking at `include/uapi/linux/i2c.h`, a message structure looks as follows:

```
| struct i2c_msg {  
|     __u16 addr;    /* slave address */  
|     __u16 flags;   /* Message flags */  
|     __u16 len;     /* msg length */  
|     __u8 *buf;     /* pointer to msg data */  
| };
```

The `i2c_msg` structure describes and characterizes an I2C message. It must contain, for each message, the client address, the number of bytes of the message, and the message payload.



msg.len is a u16. It means you must always be less than 2^{16} (64k) with your read/write buffers.

Let us have a look at the `read` function for the microchip I2C 24LC512eeprom character driver; we should understand how things really work. The full code is provided with the source of this book.

```

ssize_t
eep_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    [...]
    int _reg_addr = dev->current_pointer;
    u8 reg_addr[2];
    reg_addr[0] = (u8)(_reg_addr>> 8);
    reg_addr[1] = (u8)(_reg_addr& 0xFF);

    struct i2c_msg msg[2];
    msg[0].addr = dev->client->addr;
    msg[0].flags = 0;          /* Write */
    msg[0].len = 2;            /* Address is 2bytes coded */
    msg[0].buf = reg_addr;

    msg[1].addr = dev->client->addr;
    msg[1].flags = I2C_M_RD;   /* We need to read */
    msg[1].len = count;
    msg[1].buf = dev->data;

    if (i2c_transfer(dev->client->adapter, msg, 2) < 0)
        pr_err("ee24lc512: i2c_transfer failed\n");

    if (copy_to_user(buf, dev->data, count) != 0) {
        retval = -EIO;
        goto end_read;
    }
    [...]
}

```

msg.flags should be I2C_M_RD for a read and 0 for a write transaction. Sometimes, you may not want to create struct i2c_msg but just process simple read and write.

```

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

```

```

struct mcp23016 {

```

```

    struct i2c_client *client; struct gpio_chip chip; struct mutex lock;

```

```

};

```

```

[...]

```

```

/* This function is called when one needs to change a gpio state */

```

```

static int mcp23016_set(struct mcp23016 *mcp, unsigned offset,
intval) {

```

```

    s32 value;

```

```

    unsigned bank = offset / 8 ; u8 reg_gpio = (bank == 0) ? GP0 : GP1;
    unsigned bit = offset % 8 ;

```

```

    value = i2c_smbus_read_byte_data(mcp->client, reg_gpio); if
(value >= 0) {

```

```

        if (val)

```

```

            value |= 1 << bit; else

```



```
    value &= ~(1 << bit); return i2c_smbus_write_byte_data(mcp-  
>client, reg_gpio, value);
```

```
    } else
```

```
    return value;
```

```
}
```

```
[...]
```

Instantiating I2C devices in the board configuration file (old and depreciated way)

We must inform the kernel about which devices are physically present on the system. There are two ways to achieve that. In the DT, as we will see later in the chapter, or through the board configuration file (which is the old and depreciated way). Let us see how to do that in the board configuration file: `struct`

`i2c_board_info` is the structure used to represent an I2C device on our board. The structure is defined as follows:

```
struct i2c_board_info {  
    char type[I2C_NAME_SIZE];  
    unsigned short addr;  
    void *platform_data;  
    int irq;  
};
```

Once again, elements not relevant for us have been removed from the structure.

In the preceding structure, `type` should contain the same value as defined in the device driver in the `i2c_driver.driver.name` field. You will then need to fill an array of `i2c_board_info` and pass it as a parameter to the `i2c_register_board_info` function in the board init routine: `int i2c_register_board_info(int busnum, struct i2c_board_info const *info, unsigned len)`

Here, `busnum` is the bus number the devices sit on. This is an old and depreciated method, so I'll not go further into it in this book. Feel free to have a look at *Documentation/i2c/instantiating-devices* in the kernel sources to see how things are done.

I2C and the device tree

As we have seen in the preceding sections, in order to configure I2C devices, there are essentially two steps:

- Define and register the I2C driver
- Define and register the I2C devices

I2C devices belong to nonmemory mapped devices family in the DT, and I2C bus is an addressable bus (by addressable, I mean you can address a specific device on the bus). In this, the `reg` property in the device node represents the device address on the bus.

I2C device nodes are all children of the bus node they seat on. Each device is assigned only an address. There is no length or range involved. Standard properties one needs to declare for I2C devices are `reg`, which represents the address of the device on the bus, and the `compatible` string, which is used to match the device with a driver. For more information on addressing, you can refer to [Chapter 6, The Concept of Device Tree](#).

```
&i2c2 { /* Phandle of the bus node */
    pcf8523: rtc@68 {
        compatible = "nxp,pcf8523";
        reg = <0x68>;
    };
    eeprom: ee24lc512@55 { /* eeprom device */
        compatible = "packt,ee24lc512";
        reg = <0x55>;
    };
};
```

The preceding sample declares an HDMI EDID chip at address 0x50, on SoC's I2C bus number 2, and a **real time clock (RTC)**, at address 0x68 on the same bus.

```

/* no extra data for this device */

static const struct of_device_id foobar_of_match[] = {

    { .compatible = "packtpub,foobar-device" }, {}

};

MODULE_DEVICE_TABLE(of, foobar_of_match);

static struct i2c_driver foo_driver = {

    .driver = {

        .name = "foo", .of_match_table = of_match_ptr(foobar_of_match),
/* Only this line is added */

    },

    .probe = foo_probe, .id_table = foo_id, };

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

    const struct of_device_id *match; match =
of_match_device(mcp23s08_i2c_of_match, &client->dev); if (match)
{

    /* Device tree code goes here */

    } else {

/*

* Platform data code comes here.

```

* One can use * pdata = dev_get_platdata(&client->dev); *

* or *id*, which is a pointer on the *i2c_device_id* entry that originated * the match, in order to use *id->driver_data* to extract the device * specific data, as described in platform driver chapter.

*/

}

[...]

}

Remark

For kernel versions older than 4.10, if one looks at `drivers/i2c/i2c-core.c`, in the `i2c_device_probe()` function (for information, it is the function the kernel calls every time an I2C device is registered to the I2C core), one will see something like this: `if (!driver->probe || !driver->id_table) return -ENODEV;`

This means that even if one does not need to use the `.id_table`, it is mandatory in the driver. In fact, one can use the OF match style only, but cannot get rid of `.id_table`. Kernel developers tried to remove the need for `.id_table` and exclusively use `.of_match_table` for device matching. The patch is available at this URL: <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=c80f52847c50109ca248c22efbf71ff10553dca4>.

Nevertheless, regressions have been found and the commit was reverted. Have a look here for details: <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=661f6c1cd926c6c973e03c6b5151d161f3a666ed>. This has been fixed since kernel version `>= 4.10`. The fix looks as follows: `/* * An I2C ID table is not mandatory, if and only if, a suitable Device * Tree match table entry is supplied for the probing device. */ if (!driver->id_table && !i2c_of_match_device(dev->driver->of_match_table, client)) return -ENODEV;`

In other words, one must define both `.id_table` and `.of_match_table` for the I2C driver, otherwise your device will not be probed for kernel version 4.10 or earlier.

&i2c3 {

status = "okay"; foo-bar: foo@55 {

compatible = "packtpub,foobar-device"; reg = <55>; }; };

Putting it all together

To summarize the steps needed to write I2C client drivers, you need to:

1. Declare device ids supported by the driver. You can do that using `i2c_device_id`. If DT is supported, use `of_device_id` too.
2. Call `MODULE_DEVICE_TABLE(i2c, my_id_table)` to register your device list with the I2C core. If device tree is supported, you must call `MODULE_DEVICE_TABLE(of, your_of_match_table)` to register your device list with the OF core.
3. Write the `probe` and `remove` functions according to their respective prototypes. If needed, write power management functions too. The `probe` function must identify your device, configure it, define per-device (private) data, and register with the appropriate kernel framework. The driver's behavior depends on what you have done in the `probe` function. The `remove` function must undo everything you have done in the `probe` function (free memory and unregister from any framework).
4. Declare and fill a `struct i2c_driver` structure and set the `id_table` field with the array of ids you have created. Set `.probe` and `.remove` fields with the name of the corresponding function you have written above. In the `.driver` substructure, set the `.owner` field to `THIS_MODULE`, set the driver name, and finally, set the `.of_match_table` field with the array of `of_device_id` if DT is supported.
5. Call the `module_i2c_driver` function with your `i2c_driver` structure that you just filled above: `module_i2c_driver(serial_eeprom_i2c_driver)` in order to register your driver with the kernel.

Summary

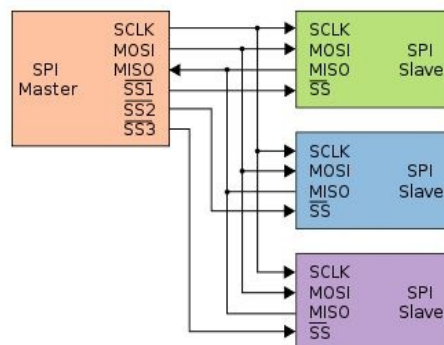
We just dealt with I2C device drivers. It is time for you to pick any I2C device on the market and write the corresponding driver, with DT support. This chapter talked about the kernel I2C core and associated API, including device tree support, to give you the necessary skills to talk with I2C devices. You should be able to write efficient `probe` functions and register with the kernel I2C core. In the next chapter, we will use skills we learned here to develop the SPI device driver.

SPI Device Drivers

Serial Peripheral Interface (SPI) is a (at least) four-wire bus--**Master Input Slave Output (MISO)**, **Master Output Slave Input (MOSI)**, **Serial Clock (SCK)**, and **Chip Select (CS)**, which is used to connect a serial flash, AD/DA converter. The master always generates the clock. Its speed can reach up to 80 MHz, even if there is no real speed limitation (much faster than I2C as well). The same for the CS line, which is always managed by the master.

Each of these signal names has a synonym:

- Whenever you sees SIMO, SDI, DI, or SDA, they refer to MOSI.
- SOMI, SDO, DO, SDA will refer to MISO.
- SCK, CLK, SCL will refer to SCK.
- \overline{SS} is the slave select line, also called CS. CSx can be used (where x is an index, CS0, CS1), EN and ENB too, meaning enable. The CS is usually an active low signal:



SPI topology (image from wikipedia) This chapter will walk through SPI driver concepts such as:

- SPI bus description
- Driver architecture and data structure descriptions
- Data sending and receiving in both half and full duplex
- Declaring SPI devices from DT
- Accessing SPI devices from user space, in both half and full duplex

The driver architecture

The required header for SPI stuff in the Linux kernel is `<linux/spi/spi.h>`. Before talking about the driver structure, let us see how SPI devices are defined in the kernel. An SPI device is represented in the kernel as an instance of `spi_device`. The instance of the driver that manages them is `struct spi_driver` structure.

The device structure

`struct spi_device` structure represents an SPI device, and is defined in `include/linux/spi/spi.h`:

```
struct spi_device {
    struct devicedev;
    struct spi_master*master;
    u32 max_speed_hz;
    u8 chip_select;
    u8 bits_per_word;
    u16 mode;
    int irq;
    [...]
    int cs_gpio;          /* chip select gpio */
};
```

Some fields that are not meaningful for us have been removed. That says, the following is the meaning of elements in the structure:

- `master`: This represents the SPI controller (bus) on which the device is connected.
- `max_speed_hz`: This is the maximum clock rate to be used with this chip (on the current board); this parameter can be changed from within the driver. You can override that parameter using `spi_transfer.speed_hz` for each transfer. We will discuss SPI transfer later.
- `chip_select`: This lets you enable the chip you need to talk to, distinguishing chips handled by the master. The `chip_select` is active low by default. This behavior can be changed in mode, by adding the `SPI_CS_HIGH` flag.
- `mode`: This defines how data should be clocked. The device driver may change this. The data clocking is **Most Significant Bit (MSB)** first, by default for each word in a transfer. This behavior can be overridden by specifying `SPI_LSB_FIRST`.
- `irq`: This represents the interrupt number (registered as device resource in your board `init` file or through the DT) you should pass to `request_irq()` to receive interrupts from this device.

A word about SPI modes; they are built using two characteristics:

- `CPOL`: This is the initial clock polarity:

- 0: Initial clock state low, and the first edge is rising
- 1: Initial clock state high, and the first state is falling
- CPHA: This is the clock phase, choosing at which edge the data will be sampled:
 - 0: Data latched at falling edge (high to low transition), whereas output changes at rising edge
 - 1: Data latched at rising edge (low to high transition), and output at falling edge

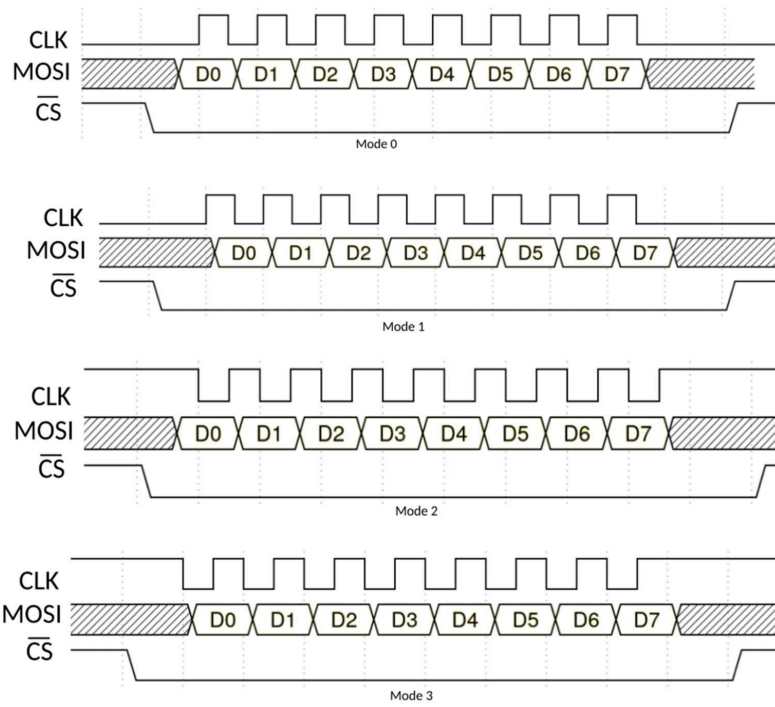
This allows for four SPI modes, which are defined in the kernel according to the following macro in `include/linux/spi/spi.h`:

```
#define SPI_CPHA 0x01
#define SPI_CPOL 0x02
```

You can then produce the following array to summarize things:

Mode	CPOL	CPHA	Kernel macro
0	0	0	<code>#define SPI_MODE_0 (0 0)</code>
1	0	1	<code>#define SPI_MODE_1 (0 SPI_CPHA)</code>
2	1	0	<code>#define SPI_MODE_2 (SPI_CPOL 0)</code>
3	1	1	<code>#define SPI_MODE_3 (SPI_CPOL SPI_CPHA)</code>

The following is the representation of each SPI mode, as defined in the preceding array. That said, only the MOSI line is represented, but the principle is the same for MISO:



Commonly used modes are `SPI_MODE_0` and `SPI_MODE_3`.

spi_driver structure

struct spi_driver represents the driver you develop to manage your SPI device. Its structure is as follows:

```
struct spi_driver {  
    const struct spi_device_id *id_table;  
    int (*probe)(struct spi_device *spi);  
    int (*remove)(struct spi_device *spi);  
    void (*shutdown)(struct spi_device *spi);  
    struct device_driver    driver;  
};
```

```

static int probe(struct spi_device *spi)

static int my_probe(struct spi_device *spi) {

    [...] /* declare your variable/structures */


    /* bits_per_word cannot be configured in platform data */

    spi->mode = SPI_MODE_0; /* SPI mode */

    spi->max_speed_hz = 20000000; /* Max clock for the device */

    spi->bits_per_word = 16; /* device bit per word */

    ret = spi_setup(spi); if (ret < 0)
    return ret;


    [...] /* Make some init */

    [...] /* Register with appropriate framework */


    return ret;

}

const struct spi_device_id *id = spi_get_device_id(spi);
my_private_data = array_chip_info[id->driver_data];

```



```

/* set the data */

void spi_set_drvdata(struct *spi_device, void *data);

/* Get the data back */

void *spi_get_drvdata(const struct *spi_device);

struct mc33880 {

    struct mutex lock; u8 bar; struct foo chip; struct spi_device *spi; };

static int mc33880_probe(struct spi_device *spi) {

    struct mc33880 *mc; [...] /* Device set up */

    mc = devm_kzalloc(&spi->dev, sizeof(struct mc33880),
GFP_KERNEL); if (!mc) return -ENOMEM;

    mutex_init(&mc->lock); spi_set_drvdata(spi, mc);

    mc->spi = spi; mc->chip.label = DRIVER_NAME, mc->chip.set =
mc33880_set;

    /* Register with appropriate framework */

    [...]

}

```

```
static int my_remove(struct spi_device *spi);

static int mc33880_remove(struct spi_device *spi) {

    struct mc33880 *mc; mc = spi_get_drvdata(spi); /* Get our data
back */

    if (!mc) return -ENODEV;

    /*

    * unregister from frameworks with which we registered in the *
probe function */

    [...]

    mutex_destroy(&mc->lock); return 0; }
```

```
static int __init foo_init(void) {  
  
    [...] /*My init code */  
  
    return spi_register_driver(&foo_driver); }  
  
module_init(foo_init);
```

```
static void __exit foo_cleanup(void) {  
  
    [...] /* My clean up code */  
  
    spi_unregister_driver(&foo_driver); }  
  
module_exit(foo_cleanup);  
  
module_spi_driver(foo_driver);
```

This will internally call `spi_register_driver` and `spi_unregister_driver`. It is exactly the same thing as what we have seen in the previous chapter.

```
struct spi_device_id {  
  
    char name[SPI_NAME_SIZE]; kernel_ulong_t driver_data; /* Data  
private to the driver */  
  
};  
  
#define ID_FOR_FOO_DEVICE 0  
  
#define ID_FOR_BAR_DEVICE 1  
  
  
static struct spi_device_id foo_idtable[] = {  
  
    { "foo", ID_FOR_FOO_DEVICE }, { "bar",  
ID_FOR_BAR_DEVICE }, { }  
  
};  
  
MODULE_DEVICE_TABLE(spi, foo_idtable);  
  
static struct spi_driver foo_driver = {  
  
    .driver = {  
  
        .name = "KBUILD_MODULE", },  
  
  
    .id_table = foo_idtable, .probe = foo_probe, .remove = foo_remove,  
};
```

```
module_spi_driver(foo_driver);
```

```

/**

* Our platform data

*/

struct my_platform_data {

    int foo; bool bar; };

static struct my_platform_data mpfd = {

    .foo = 15, .bar = true, };

static struct spi_board_info my_board_spi_board_info[] __initdata = {

    {

        /* the modalias must be same as spi device driver name */

        .modalias = "ad7887", /* Name of spi_driver for this device */

        .max_speed_hz = 1000000, /* max spi clock (SCK) speed in HZ */

        .bus_num = 0, /* Framework bus number */

        .irq = GPIO_IRQ(40), .chip_select = 3, /* Framework chip select */

        .platform_data = &mpfd, .mode = SPI_MODE_3, },{

        .modalias = "spidev", .chip_select = 0, .max_speed_hz = 1 * 1000 *

1000, .bus_num = 1, .mode = SPI_MODE_3, },

```

```
};
```

```
static int __init board_init(void) {
```

```
    [...]
```

```
    spi_register_board_info(my_board_spi_board_info,  
ARRAY_SIZE(my_board_spi_board_info)); [...]
```

```
    return 0; }
```

```
[...]
```

```
ecspi1 {
```

```
    fsl,spi-num-chipselects = <3>; cs-gpios = <&gpio5 17 0>, <&gpio5  
17 0>, <&gpio5 17 0>; pinctrl-0 = <&pinctrl_ecspi1  
&pinctrl_ecspi1_cs>; #address-cells = <1>; #size-cells = <0>;  
compatible = "fsl,imx6q-ecspi", "fsl,imx51-ecspi"; reg =  
<0x02008000 0x4000>; status = "okay";
```

```
ad7606r8_0: ad7606r8@0 {
```

```
compatible = "ad7606-8"; reg = <0>;
```

```
spi-max-frequency = <1000000>; interrupt-parent = <&gpio4>;  
interrupts = <30 0x0>; };
```

```
label: fake_spi_device@1 {
```

```
compatible = "packtpub,foobar-device"; reg = <1>;
```

```
a-string-param = "stringvalue"; spi-cs-high;
```

```
};
```

```
mcp2515can: can@2 {
```

```
compatible = "microchip,mcp2515"; reg = <2>;
```

```
spi-max-frequency = <1000000>; clocks = <&clk8m>; interrupt-  
parent = <&gpio4>; interrupts = <29 IRQ_TYPE_LEVEL_LOW>; };
```

```
};
```


There is a new property introduced in SPI device nodes: `spi-max-frequency`. It represents the maximum SPI clocking speed of the device in Hz. Whenever you access the device, the bus controller driver will ensure the clock does not cross this limit. Other properties commonly used are:

- `spi-cpol`: This is a Boolean (empty property) indicating the device requires inverse clock polarity mode. It corresponds to CPOL.
- `spi-cpha`: This is an empty property indicating the device requires shifted clock phase mode. It corresponds to CPHA.
- `spi-cs-high`: By default, SPI devices require CS low to be active. This is a Boolean property indicating the device requires CS active high.

That said, for a complete list of SPI binding elements, you can refer to *Documentation/devicetree/bindings/spi/spi-bus.txt* in the kernel sources.

```
&ecspi1 {
```

```
    status = "okay"; label: fake_spi_device@1 {
```

```
        compatible = "packtpub,foobar-device"; reg = <1>; a-string-param  
= "stringvalue"; spi-cs-high; }; };
```

```

static const struct of_device_id foobar_of_match[] = {

    { .compatible = "packtpub,foobar-device" }, { .compatible =
"packtpub,barfoo-device" }, {}

};

MODULE_DEVICE_TABLE(of, foobar_of_match);

static struct spi_driver foo_driver = {

    .driver = {

        .name = "foo", /* The following line adds Device tree */

        .of_match_table = of_match_ptr(foobar_of_match), },

    .probe = my_spi_probe, .id_table = foo_id, };

static int my_spi_probe(struct spi_device *spi) {

    const struct of_device_id *match; match =
of_match_device(of_match_ptr(foobar_of_match), &spi->dev); if
(match) {

        /* Device tree code goes here */

    } else {

        /*

        * Platform data code comes here.

        * One can use * pdata = dev_get_platdata(&spi->dev); *

```

* or *id*, which is a pointer on the *spi_device_id* entry that originated * the match, in order to use *id->driver_data* to extract the device * specific data, as described in Chapter 5, Platform Device Drivers.

*/

}

[...]

}

```

struct spi_transfer {

    const void *tx_buf; void *rx_buf; unsigned len;

    dma_addr_t tx_dma; dma_addr_t rx_dma;

    unsigned cs_change:1; unsigned tx_nbits:3; unsigned rx_nbits:3;
#define SPI_NBITS_SINGLE 0x01 /* 1bit transfer */

#define SPI_NBITS_DUAL 0x02 /* 2bits transfer */

#define SPI_NBITS_QUAD 0x04 /* 4bits transfer */

    u8 bits_per_word; u16 delay_usecs; u32 speed_hz; };

struct spi_message {

    struct list_head transfers; struct spi_device *spi; unsigned
is_dma_mapped:1; /* completion is reported through a callback */

    void (*complete)(void *context); void *context; unsigned
frame_length; unsigned actual_length; int status; };

char tx_buf[] = {

    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x40, 0x00, 0x00, 0x00,
0x00, 0x95, 0xEF, 0xBA, 0xAD,

```

0xF0, 0x0D,

};

char rx_buf[10] = {0,};

int ret;

struct spi_message single_msg;

struct spi_transfer single_xfer;

single_xfer.tx_buf = tx_buf;

single_xfer.rx_buf = rx_buf;

single_xfer.len = sizeof(tx_buff); single_xfer.bits_per_word = 8;

spi_message_init(&msg);

spi_message_add_tail(&xfer, &msg); ret = spi_sync(spi, &msg);

struct {

char buffer[10]; char cmd[2]

int foo; } data;

```
struct data my_data[3];

initialize_date(my_data, ARRAY_SIZE(my_data));

struct spi_transfer multi_xfer[3]; struct spi_message single_msg;

int ret;
```

```
multi_xfer[0].rx_buf = data[0].buffer; multi_xfer[0].len = 5;
```

```
multi_xfer[0].cs_change = 1;
```

```
/* command A */
```

```
multi_xfer[1].tx_buf = data[1].cmd; multi_xfer[1].len = 2;
```

```
multi_xfer[1].cs_change = 1;
```

```
/* command B */
```

```
multi_xfer[2].rx_buf = data[2].buffer; multi_xfer[2].len = 10;
```

```
spi_message_init(single_msg);
```

```
spi_message_add_tail(&multi_xfer[0], &single_msg);
```

```
spi_message_add_tail(&multi_xfer[1], &single_msg);
```

```
spi_message_add_tail(&multi_xfer[2], &single_msg); ret =  
spi_sync(spi, &single_msg);
```

```
int spi_read(struct spi_device *spi, void *buf, size_t len) int
```

```
spi_write(struct spi_device *spi, const void *buf, size_t len) int
```

```
spi_write_then_read(struct spi_device *spi, const void *txbuf,  
unsigned n_tx, void *rxbuf, unsigned n_rx)
```

Please have a look at `include/linux/spi/spi.h` to see the complete list. These wrappers should be used with small amounts of data.

Putting it all together

The steps needed to write SPI client drivers are as follows:

1. Declare device IDs supported by the driver. You can do that using `spi_device_id`. If DT is supported, use `of_device_id` too. You can make an exclusive use of DT.
2. Call `MODULE_DEVICE_TABLE(spi, my_id_table);` to register your device list with the SPI core. If DT is supported, you must call `MODULE_DEVICE_TABLE(of, your_of_match_table);` to register your device list with the of core.
3. Write `probe` and `remove` functions according to their respective prototypes. The `probe` function must identify your device, configure it, define per-device (private) data, configure the bus if needed (SPI mode and so on) using `spi_setup` function, and register with the appropriate kernel framework. In the `remove` function, simply undo everything done in the `probe` function.
4. Declare and fill a `struct spi_driver` structure, set the `id_table` field with the array of IDs you have created. Set `.probe` and `.remove` fields with the name of the corresponding functions you have written. In the `.driver` substructure, set the `.owner` field to `THIS_MODULE`, set the driver name, and finally set the `.of_match_table` field with the array of `of_device_id`, if the DT is supported.
5. Call `module_spi_driver` function with your `spi_driver` structure you just filled before `module_spi_driver(serial_eeprom_spi_driver);` in order to register your driver with the kernel.

```

spidev@0x00 {

    compatible = "spidev"; spi-max-frequency = <800000>; /* It
depends on your device */

    reg = <0>; /* correspond to chipselect 0 */

};

#include <stdio.h>

#include <fcntl.h>

#include <stdlib.h>

int main(int argc, char **argv) {

    int i,fd; char wr_buf[]={0xff,0x00,0x1f,0x0f}; char rd_buf[10];

    if (argc<2) {

        printf("Usage:\n%s [device]\n", argv[0]); exit(1); }

    fd = open(argv[1], O_RDWR); if (fd<=0) {

        printf("Failed to open SPI device %s\n",argv[1]); exit(1); }

    if (write(fd, wr_buf, sizeof(wr_buf)) != sizeof(wr_buf))
perror("Write Error"); if (read(fd, rd_buf, sizeof(rd_buf)) !=

```

```
sizeof(rd_buf)) perror("Read Error"); else  
    for (i = 0; i < sizeof(rd_buf); i++) printf("0x%02X ", rd_buf[i]);  
    close(fd); return 0; }
```

```
#include <stdint.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <sys/ioctl.h>

#include <linux/types.h>

#include <linux/spi/spidev.h>

static int pabort(const char *s)

{

    perror(s);

    return -1;

}


static int spi_device_setup(int fd)

{

    int mode, speed, a, b, i; int bits = 8;
```

```

/*

* spi mode: mode 0

*/

mode = SPI_MODE_0; a = ioctl(fd, SPI_IOC_WR_MODE,
&mode); /* write mode */

b = ioctl(fd, SPI_IOC_RD_MODE, &mode); /* read mode */

if ((a < 0) || (b < 0)) {

return pabort("can't set spi mode"); }


/*

* Clock max speed in Hz */

speed = 8000000; /* 8 MHz */

a = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed); /* Write
speed */

b = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed); /* Read
speed */

if ((a < 0) || (b < 0)) {

return pabort("fail to set max speed hz"); }

```

```

/*

* setting SPI to MSB first.

* Here, 0 means "not to use LSB first".

* In order to use LSB first, argument should be > 0

*/

i = 0;

a = ioctl(dev, SPI_IOC_WR_LSB_FIRST, &i); b = ioctl(dev,
SPI_IOC_RD_LSB_FIRST, &i); if ((a < 0) || (b < 0)) {

    pabort("Fail to set MSB first\n"); }


/*

* setting SPI to 8 bits per word */

bits = 8;

a = ioctl(dev, SPI_IOC_WR_BITS_PER_WORD, &bits); b =
ioctl(dev, SPI_IOC_RD_BITS_PER_WORD, &bits); if ((a < 0) || (b <
0)) {

    pabort("Fail to set bits per word\n"); }


return 0;

```

```

}

static void do_transfer(int fd)

{
    int ret;

    char txbuf[] = {0x0B, 0x02, 0xB5}; char rxbuf[3] = {0, }; char
cmd_buff = 0x9f;

    struct spi_ioc_transfer tr[2] = {

0 = {

.tx_buf = (unsigned long)&cmd_buff, .len = 1,

.cs_change = 1; /* We need CS to change */

.delay_usecs = 50, /* wait after this transfer */

.bits_per_word = 8, },

[1] = {

.tx_buf = (unsigned long)tx, .rx_buf = (unsigned long)rx, .len =
txbuf(tx), .bits_per_word = 8, },

};

    ret = ioctl(fd, SPI_IOC_MESSAGE(2), &tr); if (ret == 1){

perror("can't send spi message"); exit(1);

```

```
}
```

```
    for (ret = 0; ret < sizeof(tx); ret++) printf("%.2X ", rx[ret]);  
    printf("\n");
```

```
}
```

```
int main(int argc, char **argv)
```

```
{
```

```
    char *device = "/dev/spidev0.0"; int fd;
```

```
    int error;
```

```
    fd = open(device, O_RDWR); if (fd < 0)
```

```
        return pabort("Can't open device ");
```

```
    error = spi_device_setup(fd); if (error)
```

```
        exit (1);
```

```
    do_transfer(fd);
```

```
    close(fd);
```



```
return 0;
```

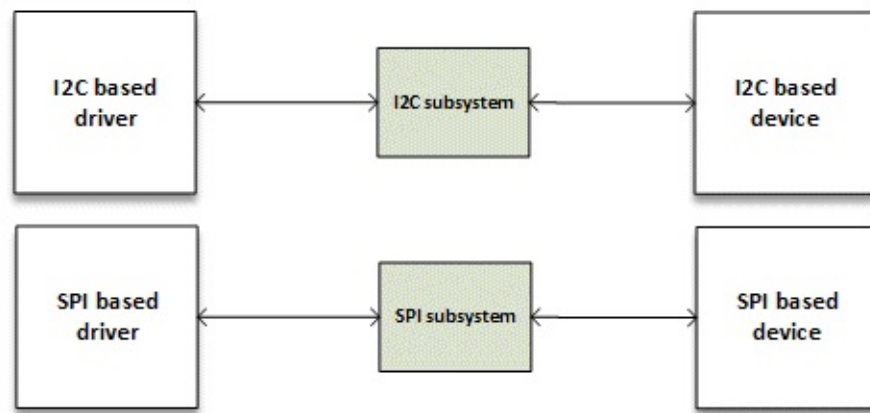
```
}
```

Summary

We just dealt with SPI drivers and now can take advantage of this faster serial (and full duplex) bus. We walked through data transfer over SPI, which is the most important section. That said, you may need more abstraction in order not to bother with SPI or I2C APIs. This is where the next chapter comes in, dealing with Regmap API, which offers a higher and unified level of abstraction, so that SPI (or I2C) commands will become transparent to you.

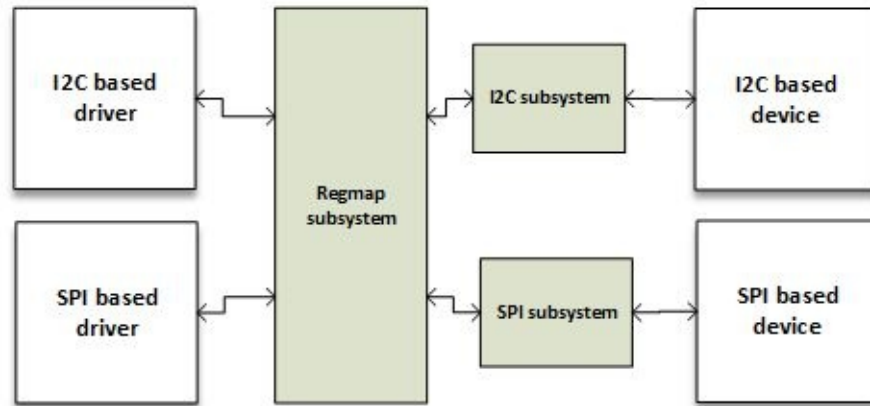
Regmap API – A Register Map Abstraction

Before the regmap API was developed, there were redundant codes for the device drivers dealing with SPI core, I2C core, or both. The principle was the same; accessing the register for read/write operations. The following figure shows how either SPI or I2C API were standalone before Regmap was introduced to kernel:



SPI and I2C subsystems before regmap

The regmap API was introduced in version 3.1 of the kernel, to factorize and unify the way kernel developers access SPI/I2C devices. It is then just a matter of how to initialize, configure a regmap, and process any read/write/modify operation fluently, whether it is SPI or I2C:



SPI and I2C subsystems after regmap

This chapter will walk through regmap framework by mean of:

- Introducing the main data structures used in by the regmap framework
- Walking through regmap configuration
- Accessing devices using the regmap API
- Introducing the regmap caching system
- Providing a complete driver that summarizes the concepts learned previously

Programming with the regmap API

The regmap API is quite simple. There are only a few structures to know. The two most important structures of this API are `struct regmap_config`, which represents the configuration of the regmap, and `struct regmap`, which is the regmap instance itself. All of the regmap data structures are defined in

`include/linux/regmap.h`.

```

struct regmap_config {

    const char *name;

    int reg_bits;

    int reg_stride;

    int pad_bits;

    int val_bits;


    bool (*writeable_reg)(struct device *dev, unsigned int reg); bool
(*readable_reg)(struct device *dev, unsigned int reg); bool
(*volatile_reg)(struct device *dev, unsigned int reg); bool
(*precious_reg)(struct device *dev, unsigned int reg); regmap_lock
lock; regmap_unlock unlock; void *lock_arg;


    int (*reg_read)(void *context, unsigned int reg, unsigned int *val);
int (*reg_write)(void *context, unsigned int reg, unsigned int val);

    bool fast_io;


    unsigned int max_register; const struct regmap_access_table
*wr_table; const struct regmap_access_table *rd_table; const struct
regmap_access_table *volatile_table; const struct
regmap_access_table *precious_table; const struct reg_default
*reg_defaults; unsigned int num_reg_defaults; enum regcache_type

```

```
cache_type; const void *reg_defaults_raw; unsigned int
num_reg_defaults_raw;
```

```
u8 read_flag_mask; u8 write_flag_mask;
```

```
bool use_single_rw; bool can_multi_write;
```

```
enum regmap_endian reg_format_endian; enum regmap_endian
val_format_endian; const struct regmap_range_cfg *ranges; unsigned
int num_ranges; }
```

```
static bool foo_writeable_register(struct device *dev, unsigned int reg)
{
```

```
    switch (reg) {
```

```
        case 0x30 ... 0x38: case 0x40 ... 0x45: case 0x50 ... 0x57: case 0x60
... 0x6e: case 0x70 ... 0x75: case 0x80 ... 0x85: case 0x90 ... 0x95:
case 0xa0 ... 0xa5: case 0xb0 ... 0xb2: return true;
```

```
        default:
```

```
            return false;
```

```
    }
```

```
}
```

```
static bool foo_volatile_register(struct device *dev, unsigned int reg) {
```

```
    switch (reg) {
```

```
        case 0x24 ... 0x29: case 0xb6 ... 0xb8: return true;
```

```
        default:
```

```
    return false;

}

}

static const struct regmap_config regmap_config = {

    .reg_bits = 8,

    .val_bits = 8,

    .max_register = LM3533_REG_MAX, .readable_reg =
lm3533_readable_register, .volatile_reg = lm3533_volatile_register,
.precious_reg = lm3533_precious_register, };
```


regmap initialization

As we said earlier, the regmap API supports both SPI and I2C protocols. Depending on the protocol you need to support in your driver, you will have to call either `regmap_init_i2c()` or `regmap_init_sp()` in the probe function. To write a generic driver, regmap is the best choice.

The regmap API is generic and homogenous. Only the initialization changes between bus types. Other functions are the same.



It is a good practice to always initialize the regmap in the probe function, and one must always fill the `regmap_config` elements prior to initializing the regmap.

Whether one allocated an I2C or SPI register map, it is freed with `regmap_exit` function: `void regmap_exit(struct regmap *map)`

This function simply release a previously allocated register map.

```
struct regmap * regmap_init_spi(struct spi_device *spi, const struct  
regmap_config);
```

```
static int foo_spi_probe(struct spi_device *client) {
```

```
    int err;
```

```
    struct regmap *my_regmap; struct regmap_config  
    bmp085_regmap_config;
```

```
    /* fill bmp085_regmap_config somewhere */
```

```
    [...]
```

```
    client->bits_per_word = 8;
```

```
    my_regmap =
```

```
    regmap_init_spi(client,&bmp085_regmap_config);
```

```
    if (IS_ERR(my_regmap)) {
```

```
        err = PTR_ERR(my_regmap); dev_err(&client->dev, "Failed to init  
regmap: %d\n", err); return err; }
```

```
    [...]
```

```
}
```

```
struct regmap * regmap_init_i2c(struct i2c_client *i2c, const struct  
regmap_config);
```

```
static int bar_i2c_probe(struct i2c_client *i2c, const struct  
i2c_device_id *id)
```

```
{
```

```
    struct my_struct * bar_struct; struct regmap_config regmap_cfg;
```

```
    /* fill regmap_cfg some where */
```

```
    [...]
```

```
    bar_struct = kzalloc(&i2c->dev, sizeof(*my_struct),  
GFP_KERNEL);
```

```
    if (!bar_struct) return -ENOMEM;
```

```
    i2c_set_clientdata(i2c, bar_struct);
```

```
    bar_struct->regmap = regmap_init_i2c(i2c, &regmap_config);
```

```
    if (IS_ERR(bar_struct->regmap)) return PTR_ERR(bar_struct->  
regmap);
```

```
    bar_struct->dev = &i2c->dev; bar_struct->irq = i2c->irq; [...]
```

```
}
```

```
int regmap_read(struct regmap *map, unsigned int reg, unsigned int
*val); int regmap_write(struct regmap *map, unsigned int reg,
unsigned int val); int regmap_update_bits(struct regmap *map,
unsigned int reg, unsigned int mask, unsigned int val);
```

- `regmap_write`: This writes data to the device. If set in `regmap_config`, `max_register`, it will be used to check if the register address you need to read from is greater or lower. If the register address passed is lower than or equal to, `max_register`, then the write operation will be performed; otherwise, the regmap core will return invalid I/O error (`-EIO`). Immediately after, the `writable_reg` callback is called. The callback must return `true` before going on to the next step. If it returns `false`, then `-EIO` is returned and the write operation stopped. If `wr_table` is set instead of `writable_reg`, then:
 - If the register address lies in `no_range`, `-EIO` is returned.
 - If the register address lies in `yes_range`, the next step is performed.
 - If the register address is present neither in `yes_range` nor `no_range`, then `-EIO` is returned and the operation is terminated.
 - If `cache_type != REGCACHE_NONE`, then cache is enabled. In this case, the cache entry is first updated, and then a write to the hardware is performed; otherwise, a no cache action is performed.
 - If `reg_write` callback is provided, it is used to perform the write operation; otherwise, the generic regmap write function will be executed.
- `regmap_read`: This reads data from the device. It works exactly like `regmap_write` with appropriate data structures (`readable_reg`, and `rd_table`). Therefore, if provided, `reg_read` is used to perform the read operation; otherwise, the generic remap

read function will be performed.

```

int regmap_update_bits(struct regmap *map, unsigned int reg,
unsigned int mask, unsigned int val)

static int _regmap_update_bits(struct regmap *map, unsigned int reg,
unsigned int mask, unsigned int val, bool *change) {

    int ret;

    unsigned int tmp, orig;

    ret = _regmap_read(map, reg, &orig); if (ret != 0)

    return ret;


    tmp = orig & ~mask; tmp |= val & mask;

    if (tmp != orig) {

    ret = _regmap_write(map, reg, tmp); *change = true; } else {

    *change = false; }


    return ret;

}

```

This way, bits you need to update must be set to 1 in mask, and the corresponding bits should be set to the value you need to give to them in val.

As an example, to set the first and third bits to 1, the mask should be

0b00000101, and the value should be 0bxxxxx1x1. To clear the seventh bit, mask must be 0b01000000 and the value should be 0bx0xxxxxx, and so on.

```

int regmap_multi_reg_write(struct regmap *map, const struct
reg_sequence *regs, int num_regs)

/**

* Register/value pairs for sequences of writes with an optional delay
in * microseconds to be applied after each write.

*

* @reg: Register address.

* @def: Register value.

* @delay_us: Delay to be applied after the register write in
microseconds */

struct reg_sequence {

    unsigned int reg; unsigned int def; unsigned int delay_us; };

static const struct reg_sequence foo_default_regs[] = {

    { FOO_REG1, 0xB8 }, { BAR_REG1, 0x00 }, {
FOO_BAR_REG1, 0x10 }, { REG_INIT, 0x00 }, { REG_POWER,
0x00 }, { REG_BLABLA, 0x00 }, };

static int probe ( ...)

{

    [...]

```



```
    ret = regmap_multi_reg_write(my_regmap, foo_default_regs,  
ARRAY_SIZE(foo_default_regs)); [...]
```

```
}
```

```
int regmap_bulk_read(struct regmap *map, unsigned int reg, void
*val, size_t val_count);
```

```
int regmap_bulk_write(struct regmap *map, unsigned int reg, const
void *val, size_t val_count);
```

Feel free to look into the regmap header file in the kernel source to see what choices you have.

```
/* Anenum of all the supported cache types */
```

```
enum regcache_type {
```

```
    REGCACHE_NONE, REGCACHE_RBTREE,  
    REGCACHE_COMPRESSED, REGCACHE_FLAT, };
```

```
/**
```

```
* Default value for a register. We use an array of structs rather * than  
a simple array as many modern devices have very sparse * register  
maps.
```

```
*
```

```
* @reg: Register address.
```

```
* @def: Register default value.
```

```
*/
```

```
struct reg_default {
```

```
    unsigned int reg; unsigned int def; };
```

```
static const struct reg_default ltc3589_reg_defaults[] = {
```

```
{ LTC3589_SCR1, 0x00 },
```

```
{ LTC3589_OVEN, 0x00 },
```

```
{ LTC3589_SCR2, 0x00 },
```

```
{ LTC3589_VCCR, 0x00 },
```

```
{ LTC3589_B1DTV1, 0x19 }, { LTC3589_B1DTV2, 0x19 }, {  
LTC3589_VRRCCR, 0xff },
```

```
{ LTC3589_B2DTV1, 0x19 }, { LTC3589_B2DTV2, 0x19 }, {  
LTC3589_B3DTV1, 0x19 }, { LTC3589_B3DTV2, 0x19 }, {  
LTC3589_L2DTV1, 0x19 }, { LTC3589_L2DTV2, 0x19 }, };
```

```
static const struct regmap_config ltc3589_regmap_config = {  
  
    .reg_bits = 8, .val_bits = 8, .writeable_reg = ltc3589_writeable_reg,  
    .readable_reg = ltc3589_readable_reg, .volatile_reg =  
    ltc3589_volatile_reg, .max_register = LTC3589_L2DTV2,  
    .reg_defaults = ltc3589_reg_defaults, .num_reg_defaults =  
    ARRAY_SIZE(ltc3589_reg_defaults), .use_single_rw = true,  
    .cache_type = REGCACHE_RBTREE, };
```

Any read operation on any one of the registers present in the array will immediately return the value in the array. However, a write operation will be performed on the device itself, and updates the affected register in the array. This way, reading the LTC3589_VRRCCR register will return 0xff; write any value in that register and it will update its entry in the array so that any new read operation will return the last written value, directly from the cache.

Putting it all together

Perform the following steps to set up regmap subsystem:

1. Set up a struct `regmap_config`, according to your device's characteristic. Set a register range if needed, default values if any, the `cache_type` if needed, and so on. If custom read/write functions are needed, pass them to `reg_read/reg_write` fields.
2. In the `probe` function, allocate a regmap using `regmap_init_i2c` OR `regmap_init_spi` depending on the bus: I2C or SPI.
3. Whenever you need to read/write from/into registers, call `regmap_[read|write]` functions.
4. When you are done with the regmap, call `regmap_exit` to free the register map allocated in `probe`.

```
/* mandatory for regmap */  
  
#include <linux/regmap.h>  
  
/* Depending on your need you should include other files */
```

```
static struct private_struct  
{  
    /* Feel free to add whatever you want here */  
    struct regmap *map; int foo;  
};
```

```
static const struct regmap_range wr_rd_range[] =  
{  
    {  
        .range_min = 0x20,  
        .range_max = 0x4F,  
    }, {  
        .range_min = 0x60,  
        .range_max = 0x7F
```

```

    },

};

struct regmap_access_table drv_wr_table =

{

    .yes_ranges = wr_rd_range, .n_yes_ranges =
ARRAY_SIZE(wr_rd_range), };

struct regmap_access_table drv_rd_table =

{

    .yes_ranges = wr_rd_range, .n_yes_ranges =
ARRAY_SIZE(wr_rd_range), };

static bool writeable_reg(struct device *dev, unsigned int reg) {

    if (reg >= 0x20 && reg <= 0x4F) return true;

    if (reg >= 0x60 && reg <= 0x7F) return true;

    return false;

}

```

```
static bool readable_reg(struct device *dev, unsigned int reg) {

    if (reg >= 0x20 && reg <= 0x4F) return true;

    if (reg >= 0x60 && reg <= 0x7F) return true;

    return false;

}
```

```
static int my_spi_drv_probe(struct spi_device *dev) {

    struct regmap_config config; struct custom_drv_private_struct
    *priv; unsigned char data;

    /* setup the regmap configuration */

    memset(&config, 0, sizeof(config)); config.reg_bits = 8;
    config.val_bits = 8; config.write_flag_mask = 0x80;
    config.max_register = 0x80; config.fast_io = true;
    config.writeable_reg = drv_writeable_reg; config.readable_reg =
    drv_readable_reg;

    /*

    * If writeable_reg and readable_reg are set, * there is no need to
    provide wr_table nor rd_table.

    * Uncomment below code only if you do not want to use *
    writeable_reg nor readable_reg.

    */
```



```

//config.wr_table = drv_wr_table; //config.rd_table = drv_rd_table;

/* allocate the private data structures */

/* priv = kzalloc */


/* Init the regmap spi configuration */

priv->map = regmap_init_spi(dev, &config); /* Use
regmap_init_i2c in case of i2c bus */


/*

* Let us write into some register * Keep in mind that, below
operation will remain same * whether you use SPI or I2C. It is an
advantage when * you use regmap.

*/

regmap_read(priv->map, 0x30, &data); [...] /* Process data */


data = 0x24;

regmap_write(priv->map, 0x23, data); /* write new value */


/* set bit 2 (starting from 0) and 6 of register 0x44 */

regmap_update_bits(priv->map, 0x44, 0b00100010, 0xFF); [...] /*

```

Lot of stuff */

return 0;

}

Summary

This chapter is all about the regmap API. How easy it is, gives you an idea about how useful and widely used it is. This chapter has told you everything you need to know about the regmap API. Now you should be able to convert any standard SPI/I2C driver into a regmap. The next chapter will cover IIO devices, a framework for an analog to digital converter. Those kinds of device always sit on top of the SPI/I2C buses. It will be a challenge for us, at the end of the next chapter, to write an IIO driver using the regmap API.

IIO Framework

Industrial I/O (IIO) is a kernel subsystem dedicated to **analogic to digital converters (ADC)** and **digital to analogic converters (DAC)**. With the growing number of sensors (measurement devices with analogue to digital, or digital to analogue, capabilities) with different code implementations, scattered over the kernel sources, gathering them became necessary. This is what IIO framework does, in a generic and homogeneous way. Jonathan Cameron and the Linux-IIO community have been developing it since 2009.

Accelerometer, Gyroscope, current/voltage measurement chips, light sensors, pressure sensors, and so on all fall into the IIO family of devices.

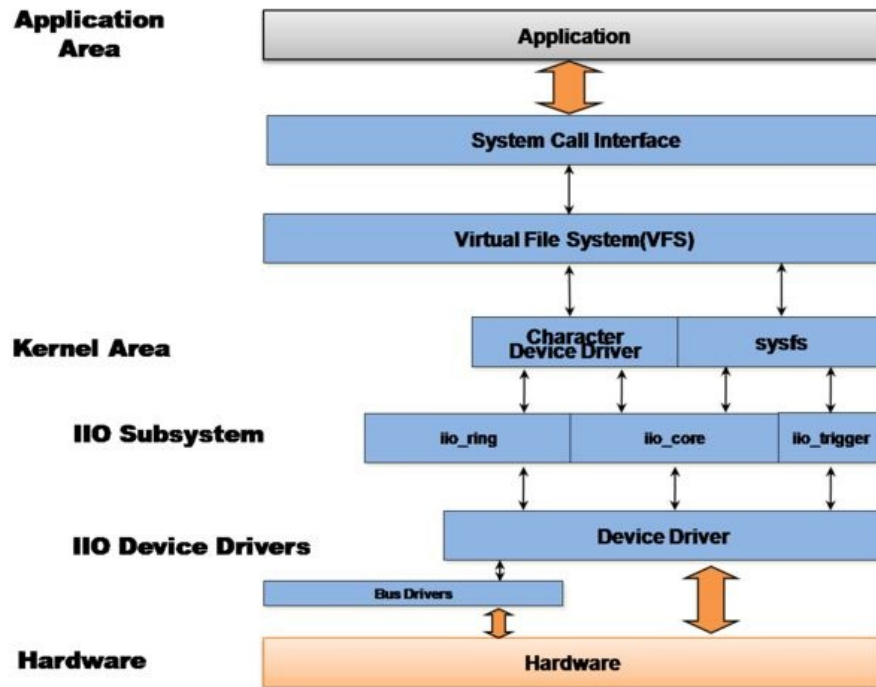
The IIO model is based on devices and channels architecture:

- Device represents the chip itself. It is the top level of the hierarchy.
- Channel represents a single acquisition line of the device. A device may have one or more channels. For example, an accelerometer is a device with three channels, one for each axis (X, Y, and Z).

The IIO chip is the physical and hardware sensor/converter. It is exposed to the user space as a character device (when triggered buffering is supported), and a **sysfs** directory entry which will contain a set of files, some of which represent the channels. A single channel is represented with a single **sysfs** file entry.

These are the two ways to interact with an IIO driver from the user space:

- `/sys/bus/iio/iio:deviceX/`: This represents the sensor along with its channels
- `/dev/iio:deviceX`: This is a character device which exports the device's events and data buffer



IIO framework architecture and layout

The preceding figure shows how the IIO framework is organized between kernel and user space. The driver manages the hardware and report processing to the IIO core, using a set of facilities and API exposed by the IIO core. The IIO subsystem then abstracts the whole underlying mechanism to the user space by means of the sysfs interface and character device, on top of which users can execute system calls.

IIO APIs are spread over several header files, listed as following:

```
#include <linux/iio/iio.h>    /* mandatory */
#include <linux/iio/sysfs.h>  /* mandatory since sysfs is used */
#include <linux/iio/events.h> /* For advanced users, to manage iio events */
#include <linux/iio/buffer.h> /* mandatory to use triggered buffers */
#include <linux/iio/trigger.h> /* Only if you implement trigger in your driver (rarely u
```

In this chapter, we will describe and handle every concepts of IIO framework, such as

- A walk through its data structure (device, channel, and so on)
- Triggered buffer support and continuous capture, along with its sysfs interface
- Exploring existing IIO triggers

- Capturing data in either one-shot mode or continuous mode
- Listing available tools that can help developers in testing their devices

IIO data structures

An IIO device is represented in the kernel as an instance of the `struct iio_dev`, and described by a `struct iio_info` structure. All of the important IIO structures are defined in `include/linux/iio/iio.h`.

```

struct iio_dev {

    [...]

    int modes;

    int currentmode; struct device dev;

    struct iio_buffer *buffer; int scan_bytes;


    const unsigned long *available_scan_masks; const unsigned long
    *active_scan_mask; bool scan_timestamp; struct iio_trigger *trig;
    struct iio_poll_func *pollfunc;


    struct iio_chan_spec const *channels; int num_channels; const char
    *name; const struct iio_info *info; const struct iio_buffer_setup_ops
    *setup_ops; struct cdev chrdev; };

struct iio_dev *devm_iio_device_alloc(struct device *dev, <br/> int
sizeof_priv)

struct iio_dev *indio_dev;

struct my_private_data *data;

indio_dev = iio_device_alloc(sizeof(*data)); if (!indio_dev)

    return -ENOMEM;

/*data is given the address of reserved momory for private data */

data = iio_priv(indio_dev);

```



```
int iio_device_register(struct iio_dev *indio_dev)
```

```
void iio_device_unregister(struct iio_dev *indio_dev)
```

```
void iio_device_free(struct iio_dev *iio_dev)
```

```
struct my_private_data *the_data = iio_priv(indio_dev);
```

```

struct iio_info {

    struct module *driver_module; const struct attribute_group *attrs;

    int (*read_raw)(struct iio_dev *indio_dev, struct iio_chan_spec
const *chan, int *val, int *val2, long mask);

    int (*write_raw)(struct iio_dev *indio_dev, struct iio_chan_spec
const *chan, int val, int val2, long mask); [...]

};

static const struct iio_info iio_dummy_info = {

    .driver_module = THIS_MODULE, .read_raw =
&iio_dummy_read_raw, .write_raw = &iio_dummy_write_raw, [...]

/*

* Provide device type specific interface functions and * constant data.

*/

indio_dev->info = &iio_dummy_info;

```

```

struct iio_chan_spec {

    enum iio_chan_type type; int channel; int channel2; unsigned long
    address; int scan_index; struct {

        char sign; u8 realbits; u8 storagebits; u8 shift; u8 repeat; enum
        iio_endian endianness; } scan_type; long info_mask_separate; long
        info_mask_shared_by_type; long info_mask_shared_by_dir; long
        info_mask_shared_by_all; const struct iio_event_spec *event_spec;
        unsigned int num_event_specs; const struct iio_chan_spec_ext_info
        *ext_info; const char *extend_name; const char *datasheet_name;
        unsigned modified:1; unsigned indexed:1; unsigned output:1;
        unsigned differential:1; };

enum iio_endian {

    IIO_CPU,

    IIO_BE,

    IIO_LE,

};

```

Channel attribute naming conventions

The attribute's name is automatically generated by the IIO core with the following pattern: `{direction}_{type}_{index}_{modifier}_{info_mask}`:

- `direction` corresponds to the attribute direction, according to the `struct iio_direction` structure in `drivers/iio/industrialio-core.c`:

```
static const char * const iio_direction[] = {  
  
    [0] = "in",  
  
    [1] = "out",  
  
};
```

- `type` corresponds to the channel type, according to the char array `const iio_chan_type_name_spec`:

```
static const char * const iio_chan_type_name_spec[] = {  
  
    [IIO_VOLTAGE] = "voltage", [IIO_CURRENT] = "current", [IIO_CAPACITANCE] = "capacitance",  
  
    [IIO_ACCEL] = "accel",  
  
    [...]
```

```
[IIO_UVINDE] = "uvindex", [IIO_ELECTRICALCONDUCTIVITY] :
```

```
[IIO_INDEX] = "index",
```

```
[IIO_GRAVITY] = "gravity", };
```

- `index` pattern depends on the channel `.indexed` field being set or not. If set, the index will be taken from the `.channel` field in order to replace the `{index}` pattern.
- `modifier` pattern depends on the channel `.modified` field being set or not. If set, the modifier will be taken from the `.channel2` field, and the `{modifier}` pattern will be replaced according to the char array struct `iio_modifier_names` structure:

```
static const char * const iio_modifier_names[] = {
```

```
[IIO_MOD_X] = "x",
```

```
[IIO_MOD_Y] = "y",
```

```
[IIO_MOD_Z] = "z",
```

```
[IIO_MOD_X_AND_Y] = "x&y", [IIO_MOD_X_AND_Z] = "x&z", [I:
```

```
[IIO_MOD_CO2] = "co2",
```

```
[IIO_MOD_VOC] = "voc",
```

```
};
```

- `info_mask` depends on the channel info mask, private or shared, indexing

value in the char array `iio_chan_info_postfix`:

```
/* relies on pairs of these shared then separate */
```

```
static const char * const iio_chan_info_postfix[] = {
```

```
    [IIO_CHAN_INFO_RAW] = "raw", [IIO_CHAN_INFO_PROCESSED] =
```

```
    [IIO_CHAN_INFO_SAMP_FREQ] = "sampling_frequency", [IIO_CHAN_INFO_SCALE] =
```

```
};
```

Distinguishing channels

You may find yourself in trouble when there are multiple data channels per channel type. The dilemma would be: how to identify them. There are two solutions for that: indexes and modifiers.

Using indexes: Given an ADC device with one channel line, indexation is not needed. Its channel definition would be:

```
static const struct iio_chan_spec adc_channels[] = {
    {
        .type = IIO_VOLTAGE,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    },
}
```

The attribute name resulting from the preceding channel described will be `in_voltage_raw`.

`/sys/bus/iio/iio:deviceX/in_voltage_raw`

Now let us say the converter has 4 or even 8 channels. How do we identify them? The solution is to use indexes. Setting the `.indexed` field to 1 will mangle the channel attribute name with the `.channel` value replacing the `{index}` pattern:

```
static const struct iio_chan_spec adc_channels[] = {
    {
        .type = IIO_VOLTAGE,
        .indexed = 1,
        .channel = 0,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    },
    {
        .type = IIO_VOLTAGE,
        .indexed = 1,
        .channel = 1,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    },
    {
        .type = IIO_VOLTAGE,
        .indexed = 1,
        .channel = 2,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    },
    {
        .type = IIO_VOLTAGE,
        .indexed = 1,
        .channel = 3,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    },
}
```

```
| }      },
```

The resulting channel attributes are:

```
/sys/bus/iio/iio:deviceX/in_voltage0_raw
/sys/bus/iio/iio:deviceX/in_voltage1_raw
/sys/bus/iio/iio:deviceX/in_voltage2_raw
/sys/bus/iio/iio:deviceX/in_voltage3_raw
```

Using modifiers: Given a light sensor with two channels—one for infrared light and one for both infrared and visible light, without index or modifier, an attribute name would be `in_intensity_raw`. Using indexes here can be error-prone, because it makes no sense to have `in_intensity0_ir_raw`, and `in_intensity1_ir_raw`. Using modifiers will help to provide meaningful attribute names. The channel's definition could look like as follows:

```
static const struct iio_chan_spec mylight_channels[] = {
    {
        .type = IIO_INTENSITY,
        .modified = 1,
        .channel2 = IIO_MOD_LIGHT_IR,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
    },
    {
        .type = IIO_INTENSITY,
        .modified = 1,
        .channel2 = IIO_MOD_LIGHT_BOTH,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
    },
    {
        .type = IIO_LIGHT,
        .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED),
        .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
    },
}
```

Resulting attributes will be:

- `/sys/bus/iio/iio:deviceX/in_intensity_ir_raw` for the channel measuring IR intensity
- `/sys/bus/iio/iio:deviceX/in_intensity_both_raw` for the channel measuring both infrared and visible light
- `/sys/bus/iio/iio:deviceX/in_illuminance_input` for the processed data
- `/sys/bus/iio/iio:deviceX/sampling_frequency` for the sampling frequency, shared by all

This is valid with accelerometer too, as we will see further on in the case study.
For now, let's summarize what we have discussed so far in a dummy IIO driver.

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

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/platform_device.h>

#include <linux/interrupt.h>

#include <linux/of.h>

#include <linux/iio/iio.h>

#include <linux/iio/sysfs.h>

#include <linux/iio/events.h>

#include <linux/iio/buffer.h>


#define FAKE_VOLTAGE_CHANNEL(num) \

{ \

    .type = IIO_VOLTAGE, \

    .indexed = 1, \

    .channel = (num), \

    .address = (num), \

    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW), \
```

```
.info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE) \
}
```

```
struct my_private_data {
    int foo;

    int bar;

    struct mutex lock; };
```

```
static int fake_read_raw(struct iio_dev *indio_dev, struct
iio_chan_spec const *channel, int *val, int *val2, long mask) {

    return 0;

}
```

```
static int fake_write_raw(struct iio_dev *indio_dev, struct
iio_chan_spec const *chan, int val, int val2, long mask) {

    return 0;

}
```

```
static const struct iio_chan_spec fake_channels[] = {
```

```
FAKE_VOLTAGE_CHANNEL(0),  
FAKE_VOLTAGE_CHANNEL(1),  
FAKE_VOLTAGE_CHANNEL(2),  
FAKE_VOLTAGE_CHANNEL(3), };
```

```
static const struct of_device_id iio_dummy_ids[] = {  
  
    { .compatible = "packt,iio-dummy-random", }, { /* sentinel */ }  
  
};
```

```
static const struct iio_info fake_iio_info = {  
  
    .read_raw = fake_read_raw, .write_raw = fake_write_raw,  
    .driver_module = THIS_MODULE, };
```

```
static int my_pdrv_probe (struct platform_device *pdev) {  
  
    struct iio_dev *indio_dev; struct my_private_data *data;  
  
    indio_dev = devm_iio_device_alloc(&pdev->dev, sizeof(*data)); if  
    (!indio_dev) {  
  
        dev_err(&pdev->dev, "iio allocation failed!\n"); return -ENOMEM;  
  
    }
```

```
data = iio_priv(indio_dev); mutex_init(&data->lock); indio_dev->dev.parent = &pdev->dev; indio_dev->info = &fake_iio_info; indio_dev->name = KBUILD_MODNAME; indio_dev->modes = INDIO_DIRECT_MODE; indio_dev->channels = fake_channels; indio_dev->num_channels = ARRAY_SIZE(fake_channels); indio_dev->available_scan_masks = 0xF;
```

```
iio_device_register(indio_dev); platform_set_drvdata(pdev, indio_dev); return 0;
```

```
}
```

```
static void my_pdrv_remove(struct platform_device *pdev) {
```

```
    struct iio_dev *indio_dev = platform_get_drvdata(pdev);  
    iio_device_unregister(indio_dev); }
```

```
static struct platform_driver mypdrv = {
```

```
    .probe = my_pdrv_probe, .remove = my_pdrv_remove, .driver = {  
        .name = "iio-dummy-random", .of_match_table =  
        of_match_ptr(iio_dummy_ids), .owner = THIS_MODULE, },  
};
```

```
module_platform_driver(mypdrv);
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");  
MODULE_LICENSE("GPL");
```

```
<strong>~# ls -l /sys/bus/iio/devices/</strong><br/>
<strong>lrwxrwxrwx 1 root root 0 Jul 31 20:26 iio:device0 ->
../../devices/platform/iio-dummy-random.0/iio:device0</strong>
<br/><strong>lrwxrwxrwx 1 root root 0 Jul 31 20:23 iio_sysfs_trigger
-> ../../devices/iio_sysfs_trigger</strong>
```

```
<strong>~# ls /sys/bus/iio/devices/iio\:device0/</strong><br/>
<strong>dev in_voltage2_raw name uevent</strong><br/>
<strong>in_voltage0_raw in_voltage3_raw power</strong><br/>
<strong>in_voltage1_raw in_voltage_scale subsystem</strong><br/>
<strong>~# cat /sys/bus/iio/devices/iio:device0/name</strong><br/>
<strong>iio_dummy_random</strong>
```

```
static int my_read_raw(struct iio_dev *indio_dev, const struct
iio_chan_spec *chan, int *val, int *val2, long mask) {

    [...]

    switch (mask) {

        case IIO_CHAN_INFO_RAW: if (iio_buffer_enabled(indio_dev))
return -EBUSY; [...]

    }
}
```

The `iio_buffer_enabled()` function simply tests if the buffer is enabled for a given IIO device.

Let us describe some important things used in the preceding section:

- `iio_buffer_setup_ops` provides buffer setup functions to be called at fixed step of the buffer configuration sequence (before/after enable/disable). If not specified, the default `iio_triggered_buffer_setup_ops` will be given to your device by the IIO core.
- `sensor_iio_pollfunc` is the trigger's top half. As with every top half, it runs in interrupt context and must do as little processing as possible. In 99% of cases, you just have to feed the timestamp associated with the capture. Once again, one can use the default IIO `iio_pollfunc_store_time` function.
- `sensor_trigger_handler` is the bottom half, which runs in a kernel thread, allowing us to do any processing including even acquiring mutex or sleep. The heavy processing should take place here. It usually reads data from the device and stores it in the internal buffer together with the timestamp recorded in the top half, and pushes it to your IIO device buffer.

A trigger is mandatory for triggered buffering. It tells the driver when to read the sample from the device and put it into the buffer. Triggered buffering is not mandatory to write IIO device drivers. One can use single shot capture through sysfs too, by reading raw attributes of the channel, which will only perform a single conversion (for the channel attribute being read). Buffer mode allows continuous conversions, thus capturing more than one channel in a single shot.

IIO trigger and sysfs (user space)

There are two locations in sysfs related to triggers:

- `/sys/bus/iio/devices/triggerY/` which is created once an IIO trigger is registered with the IIO core and corresponds to triggers with index `Y`. There is at least one attribute in the directory:
 - `name` which is the trigger name that can be later used for association with a device
- `/sys/bus/iio/devices/iio:deviceX/trigger/*` directory will be automatically created if your device supports a triggered buffer. One can associate a trigger with our device by writing the trigger's name in the `current_trigger` file.

Sysfs trigger interface

The sysfs trigger is enabled in the kernel by the `CONFIG_IIO_SYSFS_TRIGGER=y` config option, which will result in the `/sys/bus/iio/devices/iio_sysfs_trigger/` folder being automatically created, and can be used for sysfs trigger management. There will be two files in the directory, `add_trigger` and `remove_trigger`. Its driver is in `drivers/iio/trigger/iio-trig-sysfs.c`.

add_trigger file

This is used to create a new sysfs trigger. You can create a new trigger by writing a positive value (which will be used as a trigger ID) into that file. It will create the new sysfs trigger, accessible at `/sys/bus/iio/devices/triggerX`, where `x` is the trigger number.

For example:

```
| # echo 2 > add_trigger
```

This will create a new sysfs trigger, accessible at `/sys/bus/iio/devices/trigger2`. If the trigger with the specified ID is already present in the system, an invalid argument message will be returned. The sysfs trigger name pattern is `sysfsstrig{ID}`. The command `echo 2 > add_trigger` will create the trigger `/sys/bus/iio/devices/trigger2` whose name is `sysfsstrig2`:

```
| $ cat /sys/bus/iio/devices/trigger2/name  
sysfsstrig2
```

Each sysfs trigger contains at least one file: `trigger_now`. Writing `1` into that file will instruct all devices having the corresponding trigger name in their `current_trigger` to start capture, and push data into their respective buffer. Each device buffer must have its size set, and must be enabled (`echo 1 > /sys/bus/iio/devices/iio:deviceX/buffer/enable`).

remove_trigger file

To remove a trigger, the following command is used:

```
| # echo 2 > remove_trigger
```

Tying a device with a trigger

Associating a device with a given trigger consists of writing the name of the trigger to the `current_trigger` file available under the device's trigger directory. For example, let us say we need to tie a device with the trigger that has index 2:

```
| # set trigger2 as current trigger for device0  
| # echo sysfsstrig2 > /sys/bus/iio/devices/iio:device0/trigger/current_trigger
```

To detach the trigger from the device, one should write an empty string to the `current_trigger` file of the device trigger directory, shown as follows:

```
| # echo "" > iio:device0/trigger/current_trigger
```

We will see further on in the chapter a practical example dealing with the sysfs trigger for data capture.

The interrupt trigger interface

Consider the following sample:

```
static struct resource iio_irq_trigger_resources[] = {
    [0] = {
        .start = IRQ_NR_FOR_YOUR_IRQ,
        .flags = IORESOURCE_IRQ | IORESOURCE_IRQ_LOWEDGE,
    },
};

static struct platform_device iio_irq_trigger = {
    .name = "iio_interrupt_trigger",
    .num_resources = ARRAY_SIZE(iio_irq_trigger_resources),
    .resource = iio_irq_trigger_resources,
};

platform_device_register(&iio_irq_trigger);
```

Declare our IRQ trigger and it will result in the IRQ trigger standalone module being loaded. If its `probe` function succeeds, there will be a directory corresponding to the trigger. IRQ trigger names have the form `irqtrigX`, where `x` corresponds to the virtual IRQ you just passed, the one you will see in `/proc/interrupt`:

```
$ cd /sys/bus/iio/devices/trigger0/
$ cat name
```

`irqtrig85`: As we have done with other triggers, you just have to assign that trigger to your device, by writing its name into your device `current_trigger` file.

```
# echo "irqtrig85" > /sys/bus/iio/devices/iio:device0/trigger/current_trigger
```

Now, every time the interrupt will be fired, device data will be captured.



The IRQ trigger driver does not support DT yet, which is the reason why we used our board `init` file. But it does not matter; since the driver requires a resource, we can use DT without any code change.

The following is an example of device tree node declaring the IRQ trigger interface:

```
mylabel: my_trigger@0{
    compatible = "iio_interrupt_trigger";
```

```
|    interrupt-parent = <&gpio4>;  
|    interrupts = <30 0x0>;  
|};
```

The example supposes the IRQ line is the GPIO#30 that belongs to the GPIO controller node `gpio4`. This consists of using a GPIO as an interrupt source, so that whenever the GPIO changes to a given state, the interrupt is raised, thus triggering the capture.

The hrtimer trigger interface

The `hrtimer` trigger relies on the `configfs` file system (see *Documentation/iio/iio_configfs.txt* in kernel sources), which can be enabled through the `CONFIG_IIO_CONFIGFS` config option, and mounted on our system (usually under the `/config` directory):

```
| # mkdir /config  
| # mount -t configfs none /config
```

Now, loading the module `iio-trig-hrtimer` will create IIO groups accessible under `/config/iio`, allowing users to create `hrtimer` triggers under `/config/iio/triggers/hrtimer`.

For example:

```
| # create a hrtimer trigger  
| $ mkdir /config/iio/triggers/hrtimer/my_trigger_name  
| # remove the trigger  
| $ rmdir /config/iio/triggers/hrtimer/my_trigger_name
```

Each `hrtimer` trigger contains a single `sampling_frequency` attribute in the trigger directory. A full and working example is provided further in the chapter in the section *Data capture using hrtimer trigger*.

IIO buffers

The IIO buffer offers continuous data capture, where more than one data channel can be read at once. The buffer is accessible from the user space through the `/dev/iio:device` character device node. From within the trigger handler, the function used to fill the buffer is `iio_push_to_buffers_with_timestamp`. The function responsible to allocate the trigger buffer for your device is `iio_triggered_buffer_setup()`.

IIO buffer sysfs interface

An IIO buffer has an associated attributes directory under `/sys/bus/iio/iio:deviceX/buffer/*`. Here are some of the existing attributes:

- `length`: The total number of data samples (capacity) that can be stored by the buffer. This is the number of scans contained by the buffer.
- `enable`: This activates buffer capture, start the buffer capture.
- `watermark`: This attribute has been available since kernel version v4.2. It is a positive number which specifies how many scan elements a blocking read should wait for. If using `poll` for example, it will block until the watermark is reached. It makes sense only if the watermark is greater than the requested amount of reads. It does not affect non-blocking reads. One can block on poll with a timeout and read the available samples after the timeout expires, and thus have a maximum delay guarantee.

7 6 5 4 3 2 1 0

+---+---+---+---+---+---+---+---+

|D3 |D2 |D1 |D0 | X | X | X | X | (LOW byte, address 0x06) +---+---
+---+---+---+---+---+---+

7 6 5 4 3 2 1 0

+---+---+---+---+---+---+---+---+

|D11|D10|D9 |D8 |D7 |D6 |D5 |D4 | (HIGH byte, address 0x07) +---
+---+---+---+---+---+---+---+

```
<strong>$ cat  
/sys/bus/iio/devices/iio:device0/scan_elements/in_accel_y_type  
le:s12/16>>4</strong>
```

```
struct iio_chan_spec {  
  
    [...]  
  
    struct {  
  
        char sign; /* Should be 'u' or 's' as explained above */  
  
        u8 realbits; u8 storagebits; u8 shift; u8 repeat; enum iio_endian  
endianness; } scan_type; [...]  
  
    };  
  
    struct struct iio_chan_spec accel_channels[] = {  
  
        {  
  
            .type = IIO_ACCEL, .modified = 1, .channel2 = IIO_MOD_X, /*  
other stuff here */  
  
            .scan_index = 0, .scan_type = {  
  
                .sign = 's', .realbits = 12, .storagebits = 16, .shift = 4, .endianness =
```

```
IIO_LE, },
```

```
}
```

```
/* similar for Y (with channel2 = IIO_MOD_Y, scan_index = 1) *  
and Z (with channel2 = IIO_MOD_Z, scan_index = 2) axis */
```

```
}
```

```
#define BMA220_DATA_SHIFT 2

#define BMA220_DEVICE_NAME "bma220"

#define BMA220_SCALE_AVAILABLE "0.623 1.248 2.491 4.983"


#define BMA220_ACCEL_CHANNEL(index, reg, axis) { \

    .type = IIO_ACCEL, \

    .address = reg, \

    .modified = 1, \

    .channel2 = IIO_MOD_##axis, \

    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW), \

    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE), \

    .scan_index = index, \

    .scan_type = { \

        .sign = 's', \

        .realbits = 6, \

        .storagebits = 8, \

        .shift = BMA220_DATA_SHIFT, \

        .endianness = IIO_CPU, \
```

```

    }, \
}

```

```

static const struct iio_chan_spec bma220_channels[] = {

```

```

    BMA220_ACCEL_CHANNEL(0, BMA220_REG_ACCEL_X, X),
    BMA220_ACCEL_CHANNEL(1, BMA220_REG_ACCEL_Y, Y),
    BMA220_ACCEL_CHANNEL(2, BMA220_REG_ACCEL_Z, Z), };

```

```

<strong>jma@jma:~$ ls -l /sys/bus/iio/devices/iio:device0/</strong>

```

```

<span class="packt_screen">(…)<br/></span><strong># without
modifier, a channel name would have in_accel_raw (bad)</strong>

```

```

<br/><strong>-rw-r--r-- 1 root root 4096 jul 20 14:13

```

```

in_accel_scale</strong><br/><strong>-rw-r--r-- 1 root root 4096 jul
20 14:13 in_accel_x_raw</strong><br/><strong>-rw-r--r-- 1 root root
4096 jul 20 14:13 in_accel_y_raw</strong><br/><strong>-rw-r--r-- 1
root root 4096 jul 20 14:13 in_accel_z_raw</strong><span

```

```

class="packt_screen"><br/></span><span class="packt_screen">(…)</span>
</span>

```

```

<strong>$ cat

```

```

/sys/bus/iio/devices/iio:device0/scan_elements/in_accel_x_type<br/>

```

```

<span>le:s6/8>>2</span></strong>

```

```

static irqreturn_t bma220_trigger_handler(int irq, void *p) {

```

```

    int ret;

```

```

    struct iio_poll_func *pf = p; struct iio_dev *indio_dev = pf-
>indio_dev; struct bma220_data *data = iio_priv(indio_dev); struct
spi_device *spi = data->spi_device;

```

```
mutex_lock(&data->lock); data->tx_buf[0] =  
BMA220_REG_ACCEL_X | BMA220_READ_MASK; ret =  
spi_write_then_read(spi, data->tx_buf, 1, data->buffer,  
ARRAY_SIZE(bma220_channels) - 1); if (ret < 0)
```

```
goto err;
```

```
iio_push_to_buffers_with_timestamp(indio_dev, data->buffer, pf-  
>timestamp);
```

```
err:
```

```
mutex_unlock(&data->lock); iio_trigger_notify_done(indio_dev-  
>trig);
```

```
return IRQ_HANDLED;
```

```
}
```

```
static int bma220_read_raw(struct iio_dev *indio_dev, struct  
iio_chan_spec const *chan, int *val, int *val2, long mask) {
```

```
int ret;
```

```
u8 range_idx;
```

```
struct bma220_data *data = iio_priv(indio_dev);
```

```
switch (mask) {
```

```
case IIO_CHAN_INFO_RAW: /* If buffer mode enabled, do not  
process single-channel read */
```



```

if (iio_buffer_enabled(indio_dev)) return -EBUSY;

/* Else we read the channel */

ret = bma220_read_reg(data->spi_device, chan->address); if (ret <
0)

return -EINVAL;

*val = sign_extend32(ret >> BMA220_DATA_SHIFT, 5); return
IIO_VAL_INT;

case IIO_CHAN_INFO_SCALE: ret = bma220_read_reg(data-
>spi_device, BMA220_REG_RANGE); if (ret < 0)

return ret;

range_idx = ret & BMA220_RANGE_MASK; *val =
bma220_scale_table[range_idx][0]; *val2 =
bma220_scale_table[range_idx][1]; return
IIO_VAL_INT_PLUS_MICRO; }

return -EINVAL;

}

static int bma220_write_raw(struct iio_dev *indio_dev, struct
iio_chan_spec const *chan, int val, int val2, long mask) {

int i;

int ret;

```

```

int index = -1;

struct bma220_data *data = iio_priv(indio_dev);

switch (mask) {

    case IIO_CHAN_INFO_SCALE: for (i = 0; i <
ARRAY_SIZE(bma220_scale_table); i++) if (val ==
bma220_scale_table[i][0] && val2 == bma220_scale_table[i][1]) {

        index = i;

        break;

    }

    if (index < 0)

        return -EINVAL;

    mutex_lock(&data->lock); data->tx_buf[0] =
BMA220_REG_RANGE; data->tx_buf[1] = index; ret =
spi_write(data->spi_device, data->tx_buf, sizeof(data->tx_buf)); if
(ret < 0)

        dev_err(&data->spi_device->dev, "failed to set measurement
range\n"); mutex_unlock(&data->lock);

    return 0;

}

```

```

    return -EINVAL;

}

static const struct iio_info bma220_info = {

    .driver_module = THIS_MODULE, .read_raw = bma220_read_raw,
    .write_raw = bma220_write_raw, /* Only if your driver need it */

};

/*

* We provide only two mask possibility, allowing to select none or
every * channels.

*/

static const unsigned long bma220_accel_scan_masks[] = {

    BIT(AXIS_X) | BIT(AXIS_Y) | BIT(AXIS_Z),

```

0

};

```
static int bma220_probe(struct spi_device *spi) {

    int ret;

    struct iio_dev *indio_dev; struct bma220_data *data;

    indio_dev = devm_iio_device_alloc(&spi->dev, sizeof(*data)); if
(!indio_dev) {

        dev_err(&spi->dev, "iio allocation failed!\n"); return -ENOMEM;

    }

    data = iio_priv(indio_dev); data->spi_device = spi;
    spi_set_drvdata(spi, indio_dev); mutex_init(&data->lock);

    indio_dev->dev.parent = &spi->dev; indio_dev->info =
    &bma220_info; indio_dev->name = BMA220_DEVICE_NAME;
    indio_dev->modes = INDIO_DIRECT_MODE; indio_dev->channels
    = bma220_channels; indio_dev->num_channels =
    ARRAY_SIZE(bma220_channels); indio_dev->available_scan_masks
    = bma220_accel_scan_masks;

    ret = bma220_init(data->spi_device); if (ret < 0)
```

```
return ret;
```

```
/* this call will enable trigger buffer support for the device */
```

```
ret = iio_triggered_buffer_setup(indio_dev,  
iio_pollfunc_store_time, bma220_trigger_handler, NULL); if (ret < 0)  
{
```

```
dev_err(&spi->dev, "iio triggered buffer setup failed\n"); goto  
err_suspend;
```

```
}
```

```
ret = iio_device_register(indio_dev); if (ret < 0) {
```

```
dev_err(&spi->dev, "iio_device_register failed\n");  
iio_triggered_buffer_cleanup(indio_dev); goto err_suspend;
```

```
}
```

```
return 0;
```

```
err_suspend:
```

```
return bma220_deinit(spi); }
```

One can enable this driver by means of the CONFIG_BMA220 kernel

option. That said, this is available only from v4.8 onwards in kernel. The closest device one can use for this on older kernel versions is BMA180, which one can enable using the `CONFIG_BMA180` option.

IIO data access

You may have guessed that there are only two ways to access data with the IIO framework; one-shot capture through sysfs channels, or continuous mode (triggered buffer) through an IIO character device.

One-shot capture

One-shot data capture is done through sysfs interface. By reading the sysfs entry that corresponds to a channel, you'll capture only the data specific to that channel. Given a temp sensor with two channels: one for the ambient temp, and the other for the thermocouple temp: **# cd /sys/bus/iio/devices/iio:device0 # cat in_voltage3_raw 6646**

```
| # cat in_voltage_scale  
| 0.305175781
```

Processed value is obtained by multiplying the scale by the raw value.

Voltage value: $6646 * 0.305175781 = 2028.19824053$

The device datasheet says the process value is given in MV. In our case, it corresponds to 2.02819V.

Buffer data access

To get a triggered acquisition working, the trigger support must have been implemented in your driver. Then, to acquire data from within user space, one must: create a trigger, assign it, enable the ADC channels, set the dimension of the buffer, and enable it). Here is the code for this:

Capturing using the sysfs trigger

Capturing data using the sysfs trigger consists of sending a set of command few sysfs files. Let us enumerate what we should do to achieve that:

1. **Create the trigger:** Before the trigger can be assigned to any device, it should be created:

```
| # echo 0 > /sys/devices/iio_sysfs_trigger/add_trigger
```

Here, 0 corresponds to the index we need to assign to the trigger. After this command, the trigger directory will be available under `/sys/bus/iio/devices/`, as `trigger0`.

2. **Assign the trigger to the device:** A trigger is uniquely identified by its name, which we can use in order to tie device to the trigger. Since we used 0 as index, the trigger will be named `sysfsstrig0`:

```
| # echo sysfsstrig0 > /sys/bus/iio/devices/iio:device0/trigger/current_trigger
```

We could have used this command too: `cat /sys/bus/iio/devices/trigger0/name > /sys/bus/iio/devices/iio:device0/trigger/current_trigger`. That said, if the value we wrote does not correspond to an existing trigger name, nothing will happen. To make sure we really defined a trigger, we can use `cat /sys/bus/iio/devices/iio:device0/trigger/current_trigger`.

3. **Enable some scan elements:** This step consists of choosing which channels should have their data value pushed into the buffer. One should pay attention to `available_scan_masks` in the driver:

```
| # echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage4_en  
| # echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage5_en  
| # echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage6_en  
| # echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage7_en
```

4. **Setup the buffer size:** Here one should set the number of sample sets that may be held by the buffer:

```
|          # echo 100 > /sys/bus/iio/devices/iio:device0/buffer/length
```

5. **Enable the buffer:** This step consists of marking the buffer as being ready to receive pushed data:

```
|          # echo 1 > /sys/bus/iio/devices/iio:device0/buffer/enable
```

To stop the capture, we'll have to write 0 in the same file.

6. **Fire the trigger:** Launch acquisition:

```
|          # echo 1 > /sys/bus/iio/devices/trigger0/trigger_now
```

Now acquisition is done, we can:

7. Disable the buffer:

```
|          # echo 0 > /sys/bus/iio/devices/iio:device0/buffer/enable
```

8. Detach the trigger:

```
|          # echo "" > /sys/bus/iio/devices/iio:device0/trigger/current_trigger
```

9. Dump the content of our IIO character device:

```
|          # cat /dev/iio\:device0 | xxd -
```

```

<strong> # echo
/sys/kernel/config/iio/triggers/hrtimer/trigger0</strong> <strong>#
</strong> <strong>echo 50 >
/sys/bus/iio/devices/trigger0/sampling_frequency</strong> <strong>#
</strong> <strong>echo 1 >
/sys/bus/iio/devices/iio:device0/scan_elements/in_voltage4_en</strong>
<strong>#</strong> <strong>echo 1 >
/sys/bus/iio/devices/iio:device0/scan_elements/in_voltage5_en</strong>
<strong>#</strong> <strong>echo 1 >
/sys/bus/iio/devices/iio:device0/scan_elements/in_voltage6_en</strong>
<strong>#</strong> <strong>echo 1 >
/sys/bus/iio/devices/iio:device0/scan_elements/in_voltage7_en</strong>
<strong>#</strong> <strong>echo 1 >
/sys/bus/iio/devices/iio:device0/buffer/enable</strong> <strong>#
</strong> <strong>cat /dev/iio:device0 | xxd -</strong> <strong>
<span class="packt_screen">0000000: 0188 1a30 0000 0000 8312
68a8 c24f 5a14 ...0.....h..OZ.<br/></span><span
class="packt_screen"> 0000010: 0188 1a30 0000 0000 192d 98a9
c24f 5a14 ...0.....-...OZ.<br/></span><span class="packt_screen">
[...]</span></strong>

```

```

<strong>$ cat
/sys/bus/iio/devices/iio:device0/scan_elements/in_voltage_type<br/>br

```

Voltage processing: $0x188 \gg 2 = 98 * 250 = 24500 = 24.5 \text{ v}$

IIO tools

There are some useful tools you can use in order to ease and speed up your app's development with IIO devices. They are available in `tools/iio` in the kernel tree:

- `lsiiio.c`: To enumerate IIO triggers, devices, and channels
- `iio_event_monitor.c`: Monitor an IIO device's ioctl interface for IIO events
- `generic_buffer.c`: To retrieve, process, and print data received from an IIO device's buffer
- `libiio`: A powerful library developed by analog device to interface IIO devices, and available at <https://github.com/analogdevicesinc/libiio>.

Summary

By the end of this chapter, you should now be familiar with IIO framework and vocabulary. You know what channels, device, and triggers are. You can even play with your IIO device from the user space, through sysfs or character device. The time to write your own IIO driver has come. There are a lot of available existing drivers not supporting trigger buffers. You can try to add such features in one of them. In the next chapter, we will play with the most useful/used resource on a system: the memory. Be strong, the game has just started.

Kernel Memory Management

On Linux systems, every memory address is virtual. They do not point to any address in the RAM directly. Whenever one accesses a memory location, a translation mechanism is performed in order to match the corresponding physical memory.

Let us start with a short story to introduce the virtual memory concept. Given a hotel, there can be a phone in each room, having a private number. Any installed phone, of course belongs to the hotel. None of them can be joined directly from outside the hotel.

If you need to contact an occupant of a room, let us say your friend, he must have given you the hotel's switchboard number and the room number in which he stays. Once you call the switchboard and give the room number of the occupant you need to talk to, just at this moment, the receptionist redirects your call to the real private phone of the room. Only the receptionist and the room occupant know the private number mapping:

| (switchboard number + room number) <=> private (real) phone number

Every time someone in the city (or over the world) wants to contact a room occupant, he has to pass by the hotline. He needs to know the right hotline number of the hotel, and the room number. This way, switchboard number + room number = virtual address, whereas private phone number corresponds to the physical address. There are some rules related to hotels that apply on Linux as well:

Hotel	Linux
You cannot contact an occupant who has no private phone in the room. There is not even a way to attempt to do this. Your call will be ended suddenly.	You cannot access a non-existing memory in your address space. This will cause a segmentation fault.

You cannot contact an occupant who does not exist, or whose check-in the hotel is not aware of, or whose information is not found by the switchboard.	If you access unmapped memory, the CPU raises a page fault and the OS handles it.
You can't contact an occupant whose stay is over.	You cannot access freed memory. Maybe it has been allocated to another process
Many hotels may have the same brand, but located at different places, each of them having a different hotline number. If you make a mistake with the hotline number.	Different processes may have the same virtual addresses mapped in their address space, but pointing to another different physical addresses.
There is a book (or software with a database) holding the mapping between the room number and the private phone number, and consulted by the receptionist on demand.	Virtual addresses are mapped to the physical memory by page tables, which are maintained by the operating system kernel and consulted by the processor.

That is how one can imagine the virtual addresses work on a Linux system.

In this chapter, we will deal with the whole Linux memory management system covering following topics:

- Memory layout along with address translation and MMU
- Memory allocation mechanisms (page allocator, slab allocator, kmalloc allocator, and so on)
- I/O memory access
- Mapping kernel memory to user space and implementing `mmap()` callback function
- Introducing Linux caching system
- Introducing the device managed resource framework (devres)

System memory layout - kernel space and user space

Throughout this chapter, terms such as kernel space and user space will refer to their virtual address space. On Linux systems, each process owns a virtual address space. It is a kind of memory sandbox during the process life. That address space is 4 GB in size on 32-bits systems (even on a system with physical memory less than 4 GB). For each process, that 4 GB address space is split in two parts:

- User space virtual addresses
- Kernel space virtual addresses

The way the split is done depends on a special kernel configuration option, `CONFIG_PAGE_OFFSET`, which defines where the kernel addresses section starts in a process address space. The common value is `0xC0000000` by default on 32-bit systems, but this may be changed, as it is the case for i.MX6 family processors from NXP, which uses `0x80000000`. In the whole chapter, we will consider `0xC0000000` by default. This is called 3G/1G split, where the user space is given the lower 3 GB of virtual address space, and the kernel uses the upper remaining 1 GB. A typical process's virtual address space layout looks like:



Both addresses used in the kernel and the user space are virtual addresses. The difference is that accessing a kernel address needs a privileged mode. Privileged mode has extended privileges. When the CPU runs the user space side code, the

active process is said to be running in the user mode; when the CPU runs the kernel space side code, the active process is said to be running in the kernel mode.



Given an address (virtual of course), one can distinguish whether it is a kernel space or a user space address by using process layout shown above. Every address falling into 0-3 GB, comes from the user space; otherwise, it is from the kernel.

There is a reason why the kernel shares its address space with every process: because every single process at a given moment uses system calls, which will involve the kernel. Mapping the kernel's virtual memory address into each process's virtual address space allow us to avoid the cost of switching out the memory address space on each entry to (and exit from) the kernel. It is the reason why the kernel address space is permanently mapped on top of each process in order to speed up kernel access through system calls.

The memory management unit organizes memory into units of fixed size called pages. A page consists of 4,096 bytes (4 KB). Even if this size may differ on other systems, it is fixed on ARM and x86, which are architectures we are interested in:

- A memory page, virtual page, or simply page are terms one uses to refer to a fixed-length contiguous block of virtual memory. The same name `page` is used as a kernel data structure to represent a memory page.
- On the other hand, a frame (or page frame) refers to a fixed-length contiguous block of physical memory on top of which the operating system maps a memory page. Each page frame is given a number, called **page frame number (PFN)**. Given a page, one can easily get its PFN and vice versa, using the `page_to_pfn` and `pfn_to_page` macros, which will be discussed in detail in the next sections.
- A page table is the kernel and architecture data structure used to store the mapping between virtual addresses and physical addresses. The key pair `page/frame` describes a single entry in the page table. This represents a mapping.

Since a memory page is mapped to a page frame, it goes without saying that pages and page frames have the same sizes, 4 K in our case. The size of a page is

defined in the kernel through the `PAGE_SIZE` macro.



There are situations where one needs memory to be page-aligned. One says a memory is page-aligned if its address starts exactly at the beginning of a page. For example, on a 4 K page size system, 4,096, 20,480, and 409,600 are instances of page-aligned memory addresses. In other words, any memory whose address is a multiple of the system page size is said to be page-aligned.

Physical mem

Process address space +-----> +-----+

|| 3200 M |

|||

4 GB +-----+ <-----+ | HIGH MEM |

| 128 MB |||

+-----+ <-----+ ||

+-----+ <-----+ |||

| 896 MB || +--> +-----+

3 GB +-----+ <--+ +-----> +-----+

|||| 896 MB | LOW MEM

| ///// | +-----> +-----+

||

0 GB +-----+

Low memory

The first 896 MB of kernel address space constitutes the low memory region. Early at boot, the kernel permanently maps those 896 MB. Addresses that result from that mapping are called **logical addresses**. These are virtual addresses, but can be translated into physical addresses by subtracting a fixed offset, since the mapping is permanent and known in advance. Low memory match with lower bound of physical addresses. One could define low memory as being the memory for which logical addresses exist in the kernel space. Most of the kernel memory function returns low memory. In fact, to serve different purposes, kernel memory is divided into a zone. Actually, the first 16 MB of LOWMEM is reserved for DMA usage. Because of hardware limitations, the kernel cannot treat all pages as identical. We can then identify three different memory zones in the kernel space:

- `ZONE_DMA`: This contains page frames of memory below 16 MB, reserved for **Direct Memory Access (DMA)**
- `ZONE_NORMAL`: This contains page frames of memory above 16 MB and below 896 MB, for normal use
- `ZONE_HIGHMEM`: This contains page frames of memory at and above 896 MB

That says on a 512 MB system, there will be no `ZONE_HIGHMEM`, 16 MB for `ZONE_DMA`, and 496 MB for `ZONE_NORMAL`.



Another definition of logical addresses: addresses in kernel space, mapped linearly on physical addresses, which can be converted into physical addresses just with an offset, or applying a bitmask. One can convert a physical address into a logical address using the `__pa(address)` macro, and then revert with the `__va(address)` macro.

High memory

The top 128 MB of the kernel address space is called the high memory region. It is used by the kernel to temporarily map physical memory above 1 G. When physical memory above 1 GB (or more precisely, 896 MB), needs to be accessed, the kernel uses those 128 MB to create temporary mapping to its virtual address space, thus achieving the goal of being able to access all physical pages. One could define high memory as being memory for which logical addresses do not exist, and which is not mapped permanently into kernel address space. The physical memory above 896 MB is mapped on demand to the 128 MB of the HIGHMEM region.

Mapping to access high memory is created on the fly by the kernel, and destroyed when done. This makes high memory access slower. That said, the concept of high memory does not exist on the 64-bits systems, due to the huge address range (2^{64}), where the 3G/1G split does not make sense anymore.

```

struct task_struct{

    [...]

    struct mm_struct *mm, *active_mm; [...]

}

struct mm_struct {

    struct vm_area_struct *mmap; struct rb_root mm_rb; unsigned long
    mmap_base; unsigned long task_size; unsigned long highest_vm_end;
    pgd_t * pgd;

    atomic_t mm_users;

    atomic_t mm_count;

    atomic_long_t nr_ptes; #if CONFIG_PGTABLE_LEVELS > 2

    atomic_long_t nr_pmds; #endif

    int map_count;

    spinlock_t page_table_lock; struct rw_semaphore mmap_sem;
    unsigned long hiwater_rss; unsigned long hiwater_vm; unsigned long
    total_vm; unsigned long locked_vm; unsigned long pinned_vm;
    unsigned long data_vm; unsigned long exec_vm; unsigned long
    stack_vm; unsigned long def_flags; unsigned long start_code,
    end_code, start_data, end_data; unsigned long start_brk, brk,
    start_stack; unsigned long arg_start, arg_end, env_start, env_end;

    /* Architecture-specific MM context */

```

```

mm_context_t context;

unsigned long flags; struct core_state *core_state; #ifdef
CONFIG_MEMCG

/*

* "owner" points to a task that is regarded as the canonical *
user/owner of this mm. All of the following must be true in * order for
it to be changed: *

* current == mm->owner * current->mm != mm * new_owner-
>mm == mm * new_owner->alloc_lock is held */

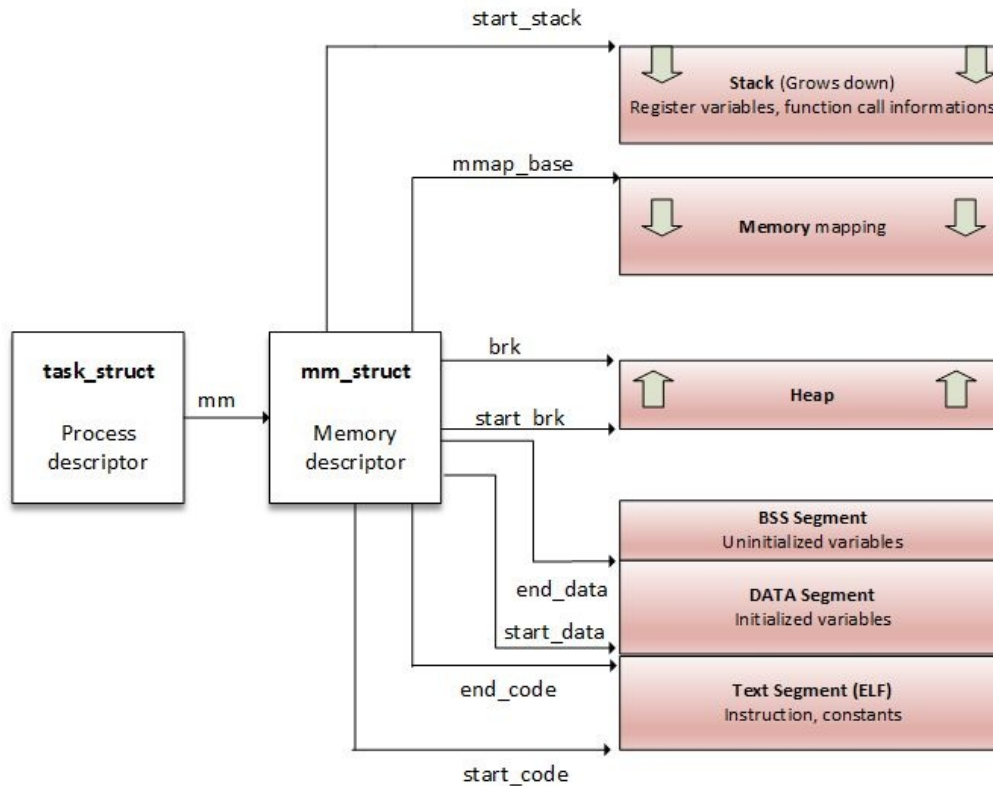
struct task_struct __rcu *owner; #endif

struct user_namespace *user_ns; /* store ref to file /proc/<pid>/exe
symlink points to */

struct file __rcu *exe_file; };

```

I intentionally removed some fields we are not interested in. There are some fields we will talk about later: `pgd` for example, which is a pointer to the process's base (first entry) level 1 table (PGD), written in the translation table base address of the CPU at context switching. Anyway, before going further, let us see the representation of a process address space:



Process memory layout From the process point of view, a memory mapping can be seen as nothing but a set of page table entries dedicated to a consecutive virtual address range. That *consecutive virtual address range* is called memory area, or **virtual memory area (VMA)**. Each memory mapping is described by a start address and length, permissions (such as whether the program can read, write, or execute from that memory), and associated resources (such as physical pages, swap pages, file contents, and so on).

A `mm_struct` has two ways to store process regions (VMA):

1. In a red-black tree, whose root element is pointed by the field `mm_struct->mm_rb`.
2. In a linked list, where the first element is pointed by the field `mm_struct->mmap`.

Virtual Memory Area (VMA)

The kernel uses virtual memory areas to keep track of the processes memory mappings, for example, a process having one VMA for its code, one VMA for each type of data, one VMA for each distinct memory mapping (if any), and so on. VMAs are processor-independent structures, with permissions and access control flags. Each VMA has a start address, a length, and their sizes are always a multiple of page size (`PAGE_SIZE`). A VMA consists of a number of pages, each of which has an entry in the page table.



Memory regions described by VMA are always virtually contiguous, not physically. One can check all VMAs associated with a process through the `/proc/<pid>/maps` file, or using the `pmap` command on a process ID.

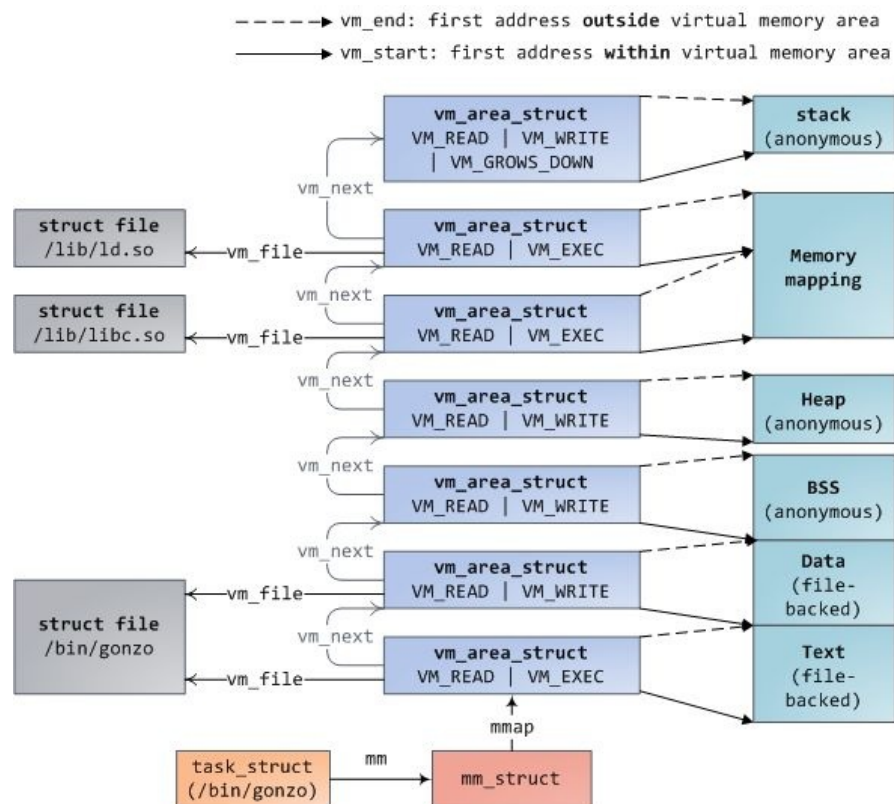


Image source: <http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/>

```
# cat /proc/1073/maps
```

```

00400000-00403000 r-xp 00000000 b3:04 6438 /usr/sbin/net-listener
00602000-00603000 rw-p 00002000 b3:04 6438 /usr/sbin/net-listener
00603000-00624000 rw-p 00000000 00:00 0 [heap]
7f0eebe4d000-7f0eebe54000 r-xp 00000000 b3:04 11717 /usr/lib/libffi.so.6.0.4
7f0eebe54000-7f0eec054000 ---p 00007000 b3:04 11717 /usr/lib/libffi.so.6.0.4
7f0eec054000-7f0eec055000 rw-p 00007000 b3:04 11717 /usr/lib/libffi.so.6.0.4
7f0eec055000-7f0eec069000 r-xp 00000000 b3:04 21629 /lib/libresolv-2.22.so
7f0eec069000-7f0eec268000 ---p 00014000 b3:04 21629 /lib/libresolv-2.22.so
[...]
7f0eee1e7000-7f0eee1e8000 rw-s 00000000 00:12 12532 /dev/shm/sem.thk-mcp-231016-sema
[...]

```

Each line in the preceding excerpt represents a VMA, and fields map the following pattern: {address (start-end)} {permissions} {offset} {device (major:minor)} {inode} {pathname (image)}:

- **address:** This represents the starting and ending address of the VMA.
- **permissions:** This describes access right of the region: *r* (read), *w* (write), and *x* (execute), including *p* (if the mapping is private) and *s* (for shared mapping).
- **offset:** In the case of file mapping (`mmap` system call), it is the offset in the file where the mapping takes place. It is `0` otherwise.
- **major:minor:** In case of file mapping, these represent the major and minor number of the devices in which the file is stored (device holding the file).
- **inode:** In the case of mapping from a file, the inode number of the mapped file.
- **pathname:** This is the name of the mapped file, or left blank otherwise. There are other region name such as `[heap]`, `[stack]`, or `[vdso]`, which stands for virtual dynamic shared object, which is a shared library mapped by the kernel into every process address space, in order to reduce performance penalties when system calls switch to kernel mode.

Each page allocated to a process belongs to an area; thus, any page that does not live in the VMA does not exist and cannot be referenced by the process.



High memory is perfect for user space because user space's virtual address must be explicitly mapped. Thus, most high memory is consumed by user applications. `__GFP_HIGHMEM` and `GFP_HIGHUSER` are the flags for requesting the allocation of (potentially) high memory. Without these flags, all kernel allocations return only low memory. There is no way to allocate contiguous physical memory from user space in Linux.

One can use the `find_vma` function to find the VMA that corresponds to a given virtual address. `find_vma` is declared in `linux/mm.h`:

```
| * Look up the first VMA which satisfies addr < vm_end, NULL if none. */  
| extern struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr);
```

This is an example:

```
| struct vm_area_struct *vma = find_vma(task->mm, 0x13000);  
| if (vma == NULL) /* Not found ? */  
|     return -EFAULT;  
| if (0x13000 >= vma->vm_end) /* Beyond the end of returned VMA ? */  
|     return -EFAULT;
```

The whole process of memory mapping can be obtained by reading files:

`/proc/<PID>/map`, `/proc/<PID>/smap`, and `/proc/<PID>/pagemap`.

Address translation and MMU

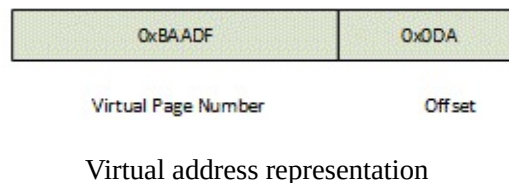
Virtual memory is a concept, an illusion given to a process so it thinks it has large and almost infinite memory, and sometimes more than the system really has. It is up to the CPU to make the conversion from virtual to physical address every time one accesses a memory location. That mechanism is called address translation, and is performed by the **Memory Management Unit (MMU)**, which is a part of the CPU.

MMU protects memory from unauthorized access. Given a process, any page that needs to be accessed must exist in one of the process VMAs, and thus, must live in the process page table (every process has its own).

Memory is organized by chunks of fixed size named **pages** for virtual memory, and **frames** for physical memory, sized 4 KB in our case. Anyway, you do not need to guess the page size of the system you write the driver for. It is defined and accessible with the `PAGE_SIZE` macro in the kernel. Remember therefore, page size is imposed by the hardware (CPU). Considering a 4 KB page sized system, bytes 0 to 4095 fall in page 0, bytes 4096-8191 fall in page 1, and so on.

The concept of page table is introduced to manage mapping between pages and frames. Pages are spread over tables, so that each PTE corresponds to a mapping between a page and a frame. Each process is then given a set of page tables to describe its whole memory space.

In order to walk through pages, each page is assigned an index (like an array), called the page number. When it comes to frame, it is PFN. This way, virtual memory addresses are composed of two parts: a page number and an offset. The offset represents the 12 less significant bits of the address, whereas 13 less significant bits represent it on 8 KB page size systems:

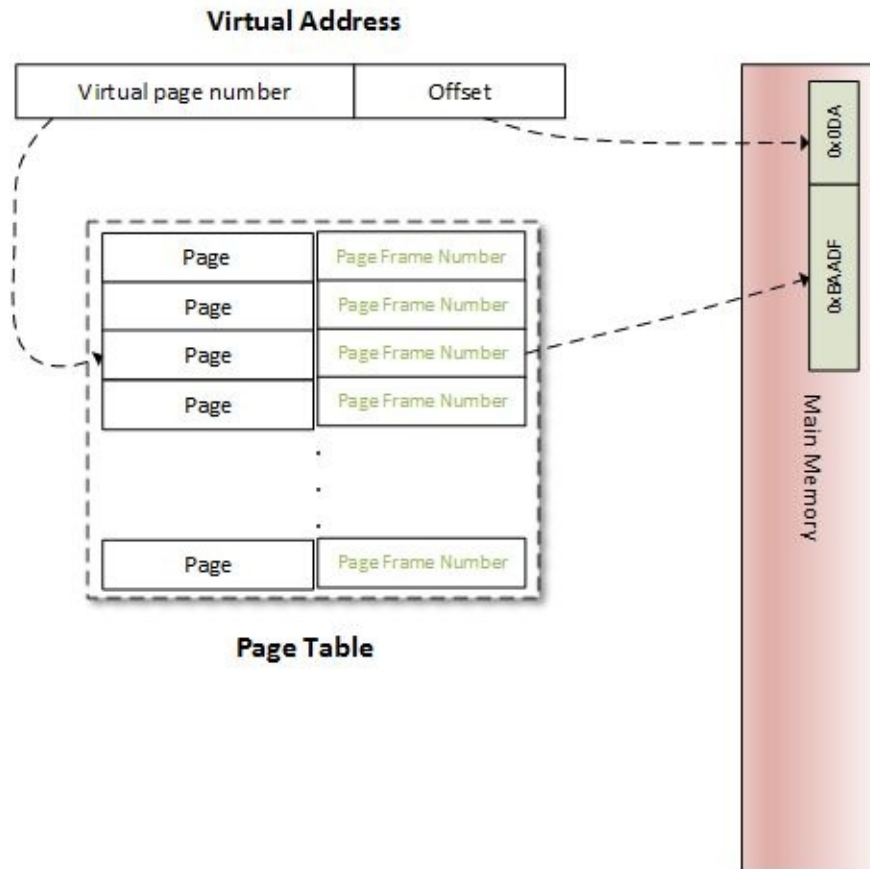


How do the OS or CPU know which physical address corresponds to a given virtual address? They use the page table as the translation table, and know that each entry's index is a virtual page number, and the value is the PFN. To access physical memory given a virtual memory, the OS first extracts the offset, the virtual page number, and then walks through the process's page tables in order to match virtual page number to physical page. Once a match occurs, it is then possible to access data into that page frame:



Address translation

The offset is used to point to the right location into the frame. Page table does not only hold mapping between physical and virtual page number, but also access control information (read/write access, privileges, and so on).



Virtual to physical address translation

The number of bits used to represent the offset is defined by the kernel macro `PAGE_SHIFT`. `PAGE_SHIFT` is the number of bits to shift one bit left to obtain the `PAGE_SIZE` value. It is also the number of bits to right-shift to convert the virtual address to the page number and the physical address to the page frame number. The following are the definitions of these macros from `/include/asm-generic/page.h` in the kernel sources:

```
#define PAGE_SHIFT      12
#ifdef __ASSEMBLY__
#define PAGE_SIZE      (1 << PAGE_SHIFT)
#else
#define PAGE_SIZE      (1UL << PAGE_SHIFT)
#endif
```

Page table is a partial solution. Let us see why. Most architecture requires 32 bits (4 bytes) to represent a PTE. Each process having its private 3 GB user space address, we need 786,432 entries to characterize and cover a process address space. It represents too much physical memory spent per process, just to

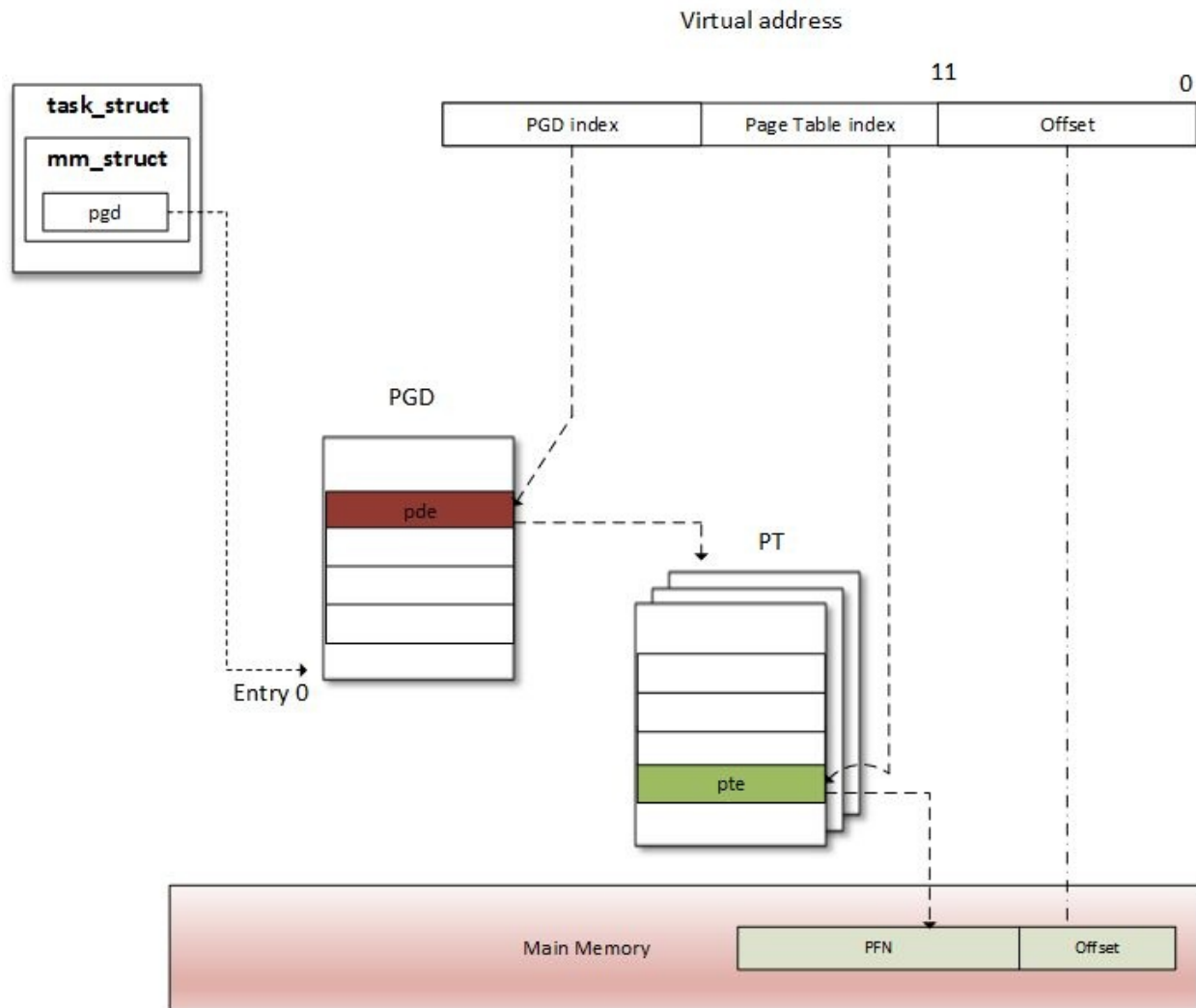
characterize the memory mappings. In fact, a process generally uses a small but scattered portion of its virtual address space. To resolve that issue, the concept of *level* is introduced. Page tables are hierarchized by level (page level). The space necessary to store a multi-level page table only depends on the virtual address space actually in use, instead of being proportional to the maximum size of the virtual address space. This way, unused memory is no longer represented, and the page table walk through time is reduced. This way, each table entry in level N will point to an entry in table of level $N+1$. Level 1 is the higher level.

Linux uses a four-level paging model:

- **Page Global Directory (PGD):** It is the first level (level 1) page table. Each entry's type is `pgd_t` in kernel (generally an `unsigned long`), and point on an entry in table at the second level. In kernel, the structure `task_struct` represents a process's description, which in turn has a member (`mm`) whose type is `mm_struct`, and that characterizes and represents the process's memory space. In the `mm_struct`, there is a processor-specific field `pgd`, which is a pointer on the first entry (entry 0) of the process's level-1 (PGD) page table. Each process has one and only one PGD, which may contain up to 1024 entries.
- **Page Upper Directory (PUD):** This exist only on architectures using four-level tables. It represent the socong level of indirection.
- **Page Middle Directory (PMD):** This is the third indirection level, and exists only on architectures using four-level tables.
- **Page Table (PTE):** Leaves of the tree. It is an array of `pte_t`, where each entry points to the physical page.



All levels are not always used. The i.MX6's MMU only supports a 2 level page table (PGD and PTE), it is the case for almost all 32-bit CPUs) In this case, PUD and PMD are simply ignored.



Two-level tables overview

You might ask how MMU is aware of the process page table. It is simple, MMU does not store any address. Instead, there is a special register in the CPU, called **page table base register (PTBR)** or **Translation Table Base Register 0 (TTBR0)**, which points to the base (entry 0) of the level-1 (top level) page table (PGD) of the process. It is exactly where the field `pgd` of `struct mm_struct` points:

```
current->mm.pgd == TTBR0.
```

At context switch (when a new process is scheduled and given the CPU), the kernel immediately configures the MMU, and updates the PTBR with the new process's `pgd`. Now when a virtual address is given to MMU, it uses the PTBR's content to locate the process's level-1 page table (PGD), and then it uses the level-1 index, extracted from the **most significant bits (MSBs)** of the virtual

address, to find the appropriate table entry, which contains a pointer to the base address of the appropriate level-2 page table. Then, from that base address, it uses the level-2 index to find the appropriate entry and so on until it reaches the PTE. ARM architecture (i.MX6 in our case) has a 2-level page table. In this case, the level-2 entry is a PTE, and points to the physical page (PFN). Only the physical page is found at this step. To access the exact memory location in the page, the MMU extracts the memory offset, also part of the virtual address, and points on the same offset in the physical page.

When a process needs to read from or write into a memory location (of course we're talking about virtual memory), the MMU performs a translation into that process's page table, to find the right entry (PTE). The virtual page number is extracted (from the virtual address) and used by the processor as an index into the processes page table to retrieve its page table entry. If there is a valid page table entry at that offset, the processor takes the page frame number from this entry. If not, it means the process accessed an unmapped area of its virtual memory. A page fault is then raised and the OS should handle it.

In the real world, address translation requires a page table walk, and it is not always a one-shot operation. There are at least as many memory accesses as there are table levels. A four-level page table would require four memory accesses. In other words, every virtual access would result in five physical memory accesses. The virtual memory concept would be useless if its access were four times slower than a physical access. Fortunately, SoC manufacturers worked hard to find a clever trick to address this performance issue: modern CPUs use a small associative and very fast memory called **translation lookaside buffer (TLB)**, in order to cache the PTEs of recently accessed virtual pages.

Page look up and TLB

Before the MMU proceeds to address translation, there is another step involved. As there is a cache for recently accessed data, there is also a cache for recently translated addresses. As a data cache speeds up the data accessing process, TLB speeds up virtual address translation (yes, address translation is a tricky task. It is content-addressable memory, abbreviated (**CAM**), where the key is the virtual address and the value is the physical address. In other words, the TLB is a cache for the MMU. At each memory access, the MMU first checks for recently used pages in the TLB, which contains a few of the virtual address ranges to which physical pages are currently assigned.

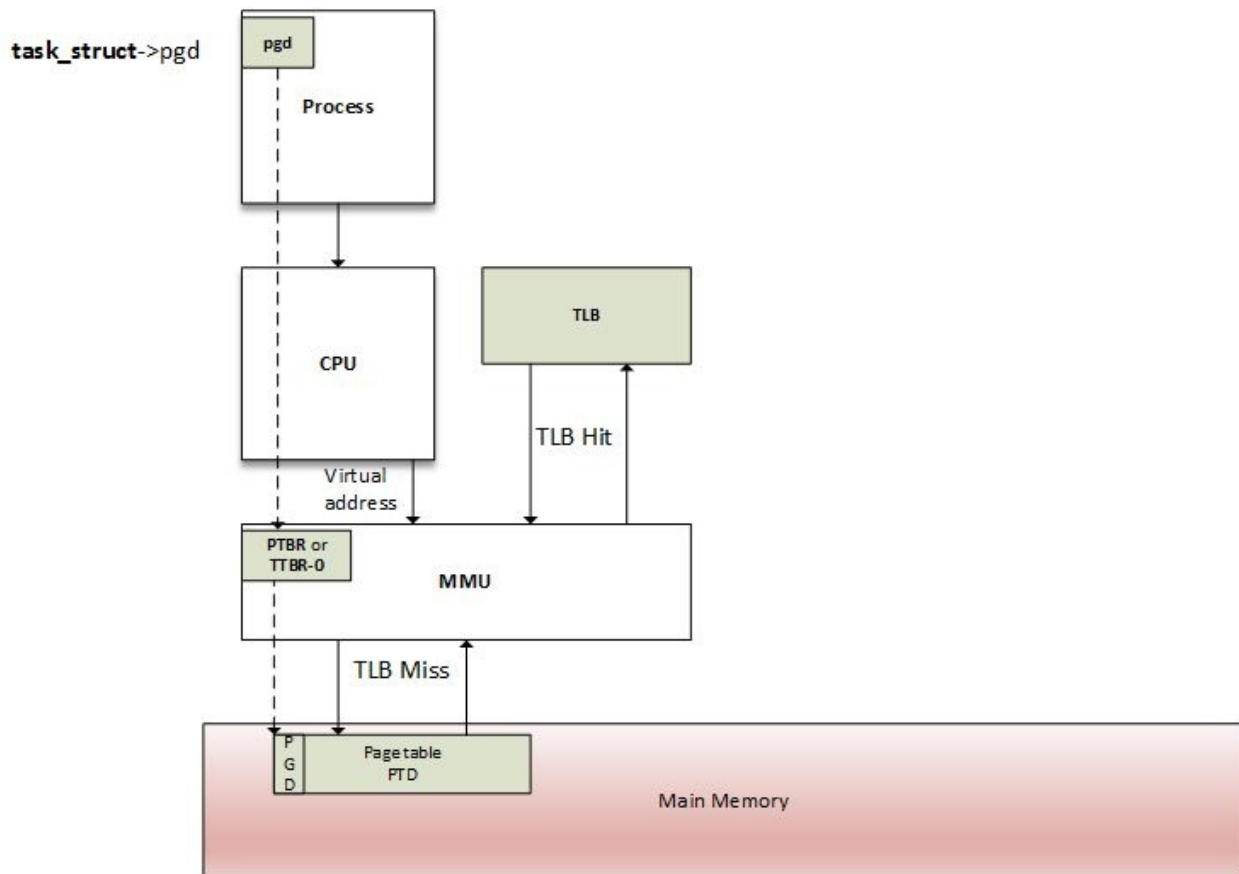
How does TLB work

On a virtual memory access, the CPU walks through the TLB trying to find the virtual page number of the page that is being accessed. This step is called TLB lookup. When a TLB entry is found (a match occurred), one says there is a **TLB hit** and the CPU just keeps running and uses the PFN found in the TLB entry to calculate the target physical address. There is no page fault when a TLB hit occurs. As one can see, as long as a translation can be found in the TLB, virtual memory access will be as fast as a physical access. If no TLB entry is found (no match occurred), one says there is a **TLB miss**.

On a TLB miss event, there are two possibilities, depending on the processor type, TLB miss events can be handled by the software, or by the hardware, through the MMU:

- **Software handling:** The CPU raises a TLB miss interruption, caught by the OS. The OS then walks through the process's page table to find the right PTE. If there is a matching and valid entry, then the CPU installs the new translation in the TLB. Otherwise, the page fault handler is executed.
- **Hardware handling:** It is up to the CPU (the MMU in fact) to walk through the process's page table in hardware. If there is a matching and valid entry, the CPU adds the new translation in the TLB. Otherwise, the CPU raises a page fault interruption, handled by the OS.

In both cases, the page fault handler is the same: the `do_page_fault()` function is executed, which is architecture-dependent. For ARM, the `do_page_fault` is defined in `arch/arm/mm/fault.c`:



MMU and TLB walkthrough process

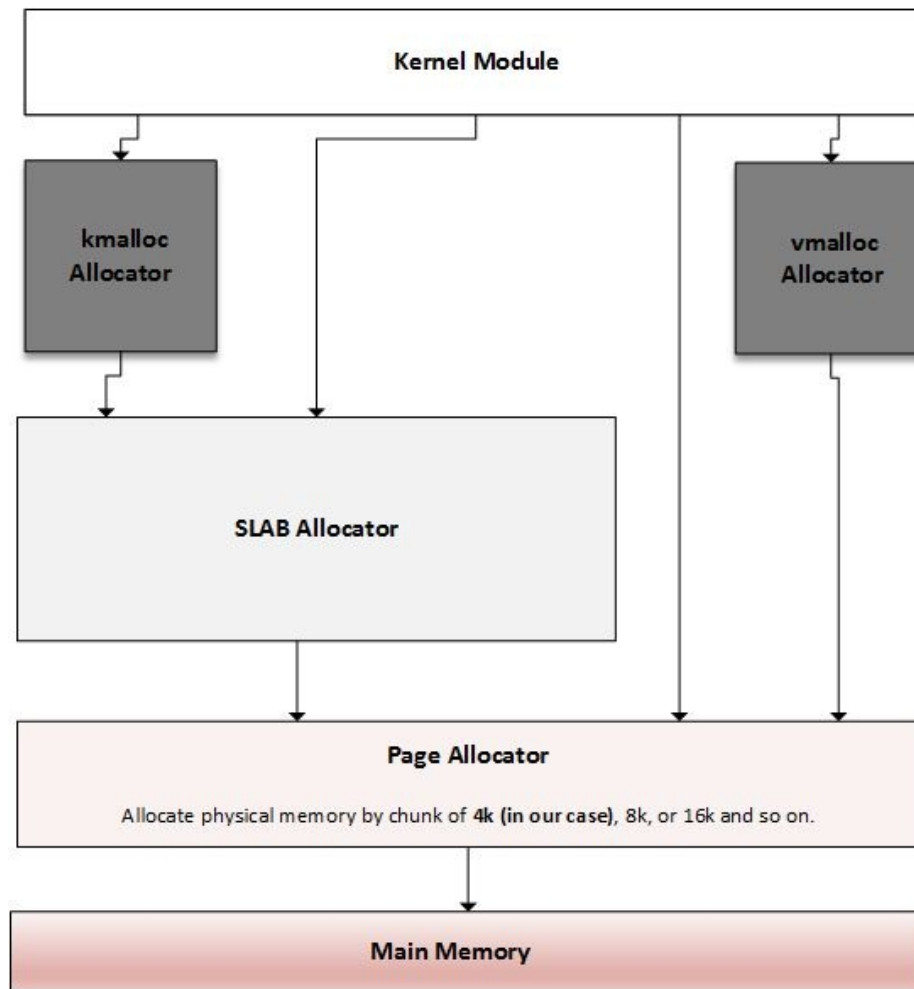


Page table and Page directory entries are architecture-dependent. It is up to the Operating system to ensure that the structure of the table corresponds to a structure recognized by the MMU. On the ARM processor, you must write the location of the translation table in CP15 (coprocessor 15) register c2, and then enable the caches and the MMU by writing to the CP15 register c1. Have a look at both <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABHJIBH.htm> and <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0433c/CIHFDBEJ.html> for detailed information.

Memory allocation mechanism

Let us look at the following figure, showing us different memory allocators existing on a Linux-based system, and discuss it later:

Inspired from: <http://free-electrons.com/doc/training/linux-kernel/linux-kernel-slides.pdf>.



Overview of kernel memory allocator

There is an allocation mechanism to satisfy any kind of memory request. Depending on what you need memory for, you can choose the one closer to your goal. The main allocator is the **Page Allocator**, which only works with pages (a page being the smallest memory unit it can deliver). Then comes the **SLAB**

Allocator that is built on top of the page allocator, getting pages from it and returning smaller memory entities (by mean of slabs and caches). This is the allocator on which the **kmalloc Allocator** relies.

Page allocator

Page allocator is the low-level allocator on the Linux system, the one on which other allocators rely on. System's physical memory is made up of fixed-size blocks (called page frames). A page frame is represented in the kernel as an instance of the `struct page` structure. A page is the smallest unit of memory that the OS will give to any memory request at low level.


```

unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
{
    struct page *page;

    /*
     * __get_free_pages() returns a 32-bit address, which cannot
     represent * a highmem page */

    VM_BUG_ON((gfp_mask & __GFP_HIGHMEM) != 0);

    page = alloc_pages(gfp_mask, order); if (!page) return 0; return
(unsigned long) page_address(page); }

```

The maximum number of pages one can allocate is 1024. It means that on a 4 Kb sized system, you can allocate up to $1024 \times 4 \text{ Kb} = 4 \text{ MB}$ at most. It is the same for `kmalloc`.

Conversion functions

The `page_to_virt()` function is used to convert the struct page (as returned by `alloc_pages()` for example) into the kernel address. `virt_to_page()` takes a kernel virtual address and returns its associated struct page instance (as if it was allocated using the `alloc_pages()` function). Both `virt_to_page()` and `page_to_virt()` are defined in `<asm/page.h>`:

```
| struct page *virt_to_page(void *kaddr);  
| void *page_to_virt(struct page *pg)
```

The macro `page_address()` can be used to return the virtual address that corresponds to the beginning address (the logical address of course) of a struct page instance:

```
| void *page_address(const struct page *page)
```

We can see how it is used in the `get_zeroed_page()` function:

```
| unsigned long get_zeroed_page(unsigned int gfp_mask)  
| {  
|     struct page * page;  
  
|     page = alloc_pages(gfp_mask, 0);  
|     if (page) {  
|         void *address = page_address(page);  
|         clear_page(address);  
|         return (unsigned long) address;  
|     }  
|     return 0;  
| }
```

`__free_pages()` and `free_pages()` can be mixed. The main difference between them is that `free_page()` takes a virtual address as a parameter, whereas `__free_page()` takes a struct page structure.

Slab allocator

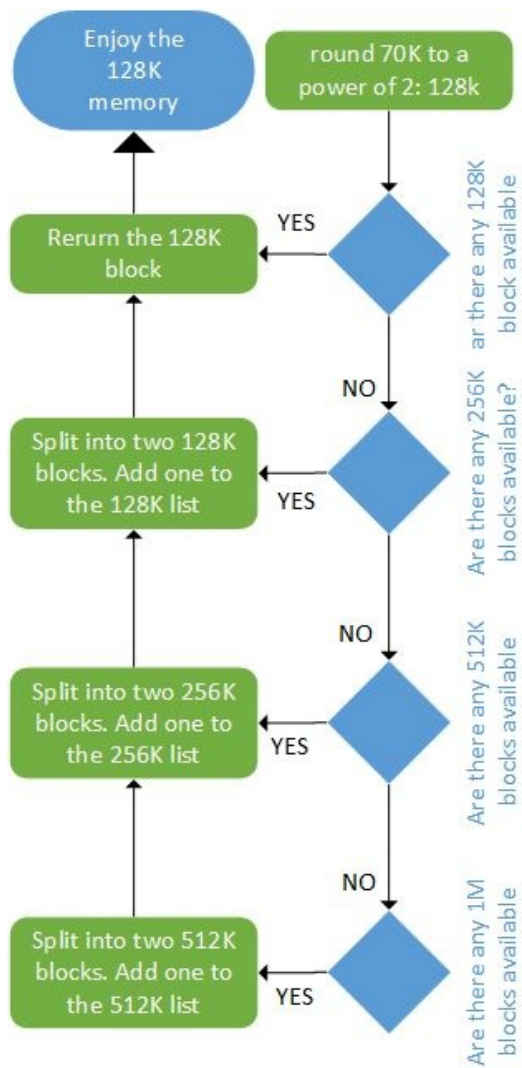
Slab allocator is the one on which `kmalloc()` relies. Its main purpose is to eliminate the fragmentation caused by memory (de)allocation that would be caused by the buddy system in the case of small size memory allocation, and speed up memory allocation for commonly used objects.

The buddy algorithm

To allocate memory, the requested size is round up to a power of two, and the buddy allocator searches the appropriate list. If no entries exist on the requested list, an entry from the next upper list (which has blocks of twice the size of the previous list) is split into two halves (called **buddies**). The allocator uses the first half, while the other is added to the next list down. This is a recursive approach, which stops when either the buddy allocator successfully finds a block which we can be split, or reaches the largest size of block and there are no free blocks available.

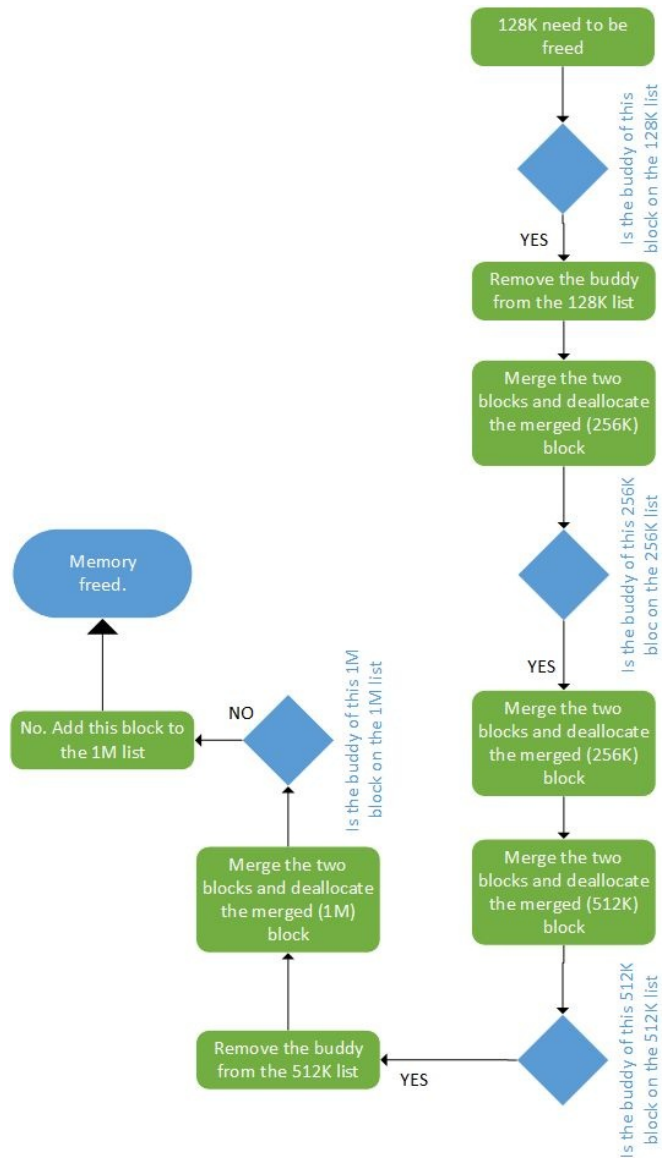
The following case study is heavily inspired from http://dysphoria.net/OperatingSystems1/4_allocation_buddy_system.html. For example, if the minimum allocation size is 1 KB, and the memory size is 1 MB, the buddy allocator will create an empty list for 1 KB holes, empty list for 2 KB holes, one for 4 KB holes, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, and one list for 1 MB holes. All of them are initially empty, except for the 1 MB list which has only one hole.

Now let us imagine a scenario where we want to allocate a **70K** block. The buddy allocator will round it up to **128K**, and end up splitting the 1 MB into two **512K** blocks, then **256K**, and finally **128K**, then it will allocate one of the **128K** blocks to the user. The following are schemes that summarize this scenario:



Allocation using buddy algorithm

The deallocation is as fast as allocation. The following figure summarize the



deallocation algorithm:

Deallocation using buddy algorithm

A journey into the slab allocator

Before we introduce the slab allocator, let us define some terms it uses:

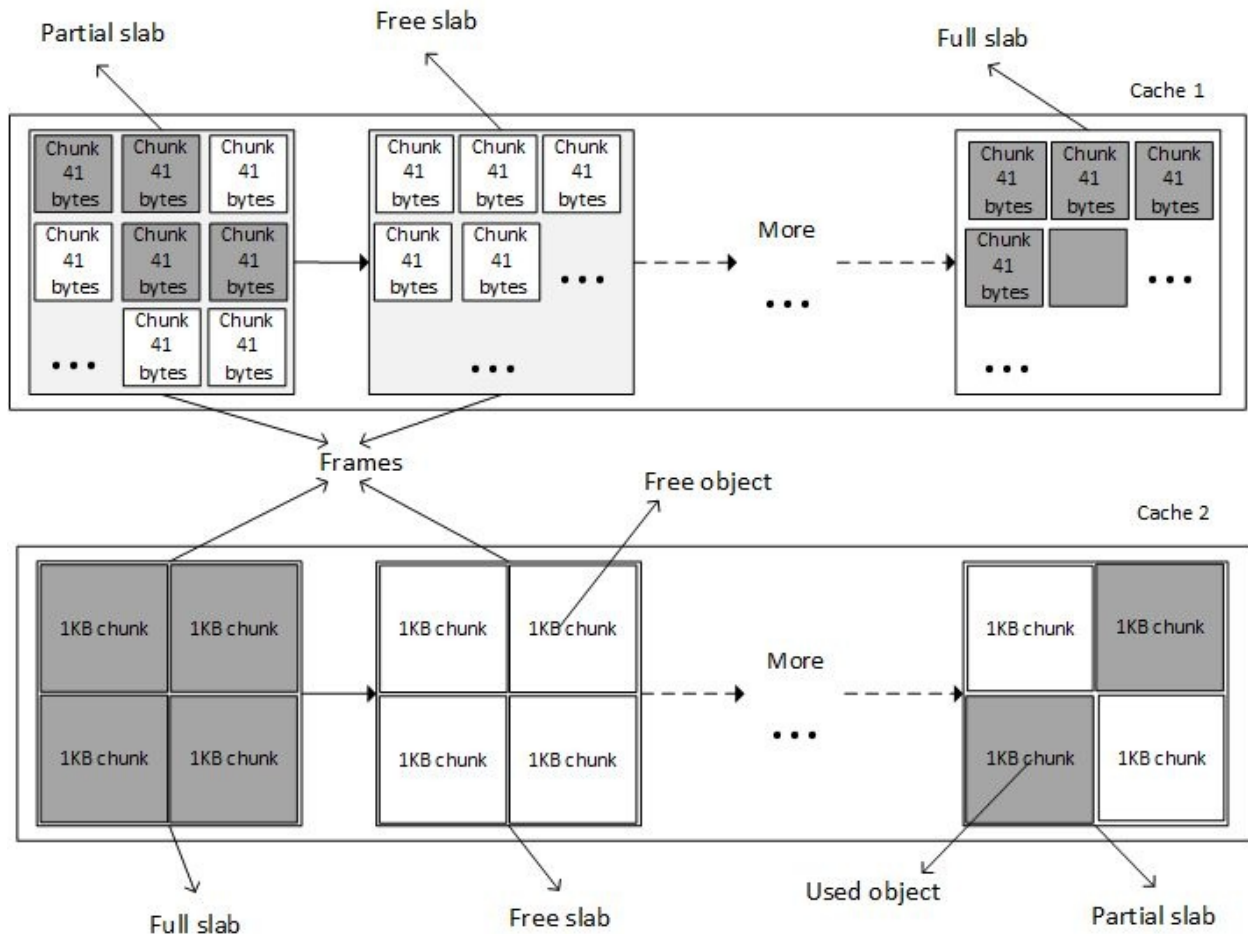
- **Slab:** This is a contiguous piece of physical memory made of several page frames. Each slab is divided into equal chunks of the same size, used to store specific types of kernel object, such as inodes, mutexes, and so on. Each slab is then an array of objects.
- **Cache:** It is made of one or more slabs in a linked list, and they are represented in the kernel as instances of the `struct kmem_cache_t` structure. The cache only stores objects of the same type (for example, inodes only, or only address space structures)

Slabs may be in one of the following states:

- **Empty:** This is where all objects (chunks) on the slab are marked as free
- **Partial:** Both used and free objects exist in the slab
- **Full:** All objects on the slab are marked as used

It is up to the memory allocator to build caches. Initially, each slab is marked as empty. When one (code) allocates memory for a kernel object, the system looks for a free location for that object on a partial/free slab in a cache for that type of object. If not found, the system allocates a new slab and adds it into the cache. The new object gets allocated from this slab, and the slab is marked as **partial**. When the code is done with the memory (memory freed), the object is simply returned to the slab cache in its initialized state.

It is the reason why the kernel also provides helper functions to obtain zeroed initialized memory, in order to get rid of the previous content. The slab keeps a reference count of how many of its objects are being used, so that when all slabs in a cache are full and another object is requested, the slab allocator is responsible for adding new slabs:



Slab cache overview

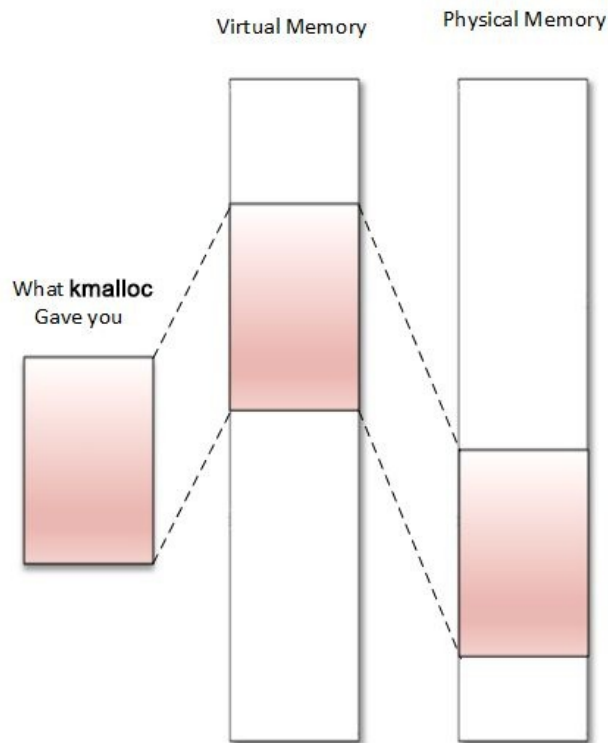
It is a bit like creating a per-object allocator. The system allocate one cache per type of object, and only objects of the same type can be stored in a cache (For example, only `task_struct` structure).

There are different kinds of slab allocator in the kernel, depending on whether or not one needs compactness, cache-friendliness, or raw speed:

- The **SLOB**, which is as compact as possible
- The **SLAB**, which is as cache-friendly as possible
- The **SLUB**, which is quite simple and requires fewer instruction cost counts

kmalloc family allocation

`kmalloc` is a kernel memory allocation function, such as `malloc()` in user space. Memory returned by `kmalloc` is contiguous in physical memory and in virtual memory:



The `kmalloc` allocator is the general and higher-level memory allocator in the kernel, which relies on the SLAB allocator. Memory returned from `kmalloc` has a kernel logical address because it is allocated from the `LOW_MEM` region, unless `HIGH_MEM` is specified. It is declared in `<linux/slab.h>`, which is the header to include when using `kmalloc` in your driver. The following is the prototype:

```
| void *kmalloc(size_t size, int flags);
```

`size` specifies the size of the memory to be allocated (in bytes). `flag` determines how and where memory should be allocated. Available flags are the same as the page allocator (`GFP_KERNEL`, `GFP_ATOMIC`, `GFP_DMA`, and so on).

- `GFP_KERNEL`: This is the standard flag. We cannot use this flag in the interrupt

handler because its code may sleep. It always returns memory from `LOWMEM` zone (hence a logical address).

- `GFP_ATOMIC`: This guarantees the atomicity of the allocation. The only flag to use when we are in the interrupt context. Please do not abuse this, since it uses an emergency pool of memory.
- `GFP_USER`: This allocates memory to a user space process. Memory is then distinct and separated from that allocated to the kernel.
- `GFP_HIGHUSER`: This allocates memory from `HIGH_MEMORY_ZONE`
- `GFP_DMA`: This allocates memory from `DMA_ZONE`.

On successful allocation of memory, `kmalloc` returns the virtual address of the chunk allocated, guaranteed to be physically contiguous. On error, it returns `NULL`.

`Kmalloc` relies on `SLAB` caches when allocating small size memories. In this case, the kernel rounds the allocated area size up to the size of the smallest `SLAB` cache in which it can fit. Always use it as your default memory allocator. In architectures used in this book (ARM and x86), the maximum size per allocation is 4 MB, and 128 MB for total allocations. Have a look at <https://kaiwante.ch.wordpress.com/2011/08/17/kmalloc-and-vmalloc-linux-kernel-memory-allocation-api-limits/>.

The `kfree` function is used to free the memory allocated by `kmalloc`. The following is the prototype of `kfree()`:

```
| void kfree(const void *ptr)
```

Let us see an example:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/mm.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("John Madieu");

void *ptr;

static int
alloc_init(void)
{
    size_t size = 1024; /* allocate 1024 bytes */
    ptr = kmalloc(size, GFP_KERNEL);
    if(!ptr) {
        /* handle error */
        pr_err("memory allocation failed\n");
        return -ENOMEM;
    }
}
```

```

    }
    else
        pr_info("Memory allocated successfully\n");
    return 0;
}

static void alloc_exit(void)
{
    kfree(ptr);
    pr_info("Memory freed\n");
}

module_init(alloc_init);
module_exit(alloc_exit);

```

Other family-like functions are:

```

void kcalloc(size_t size, gfp_t flags);
void kzfree(const void *p);
void *kcalloc(size_t n, size_t size, gfp_t flags);
void *krealloc(const void *p, size_t new_size, gfp_t flags);

```

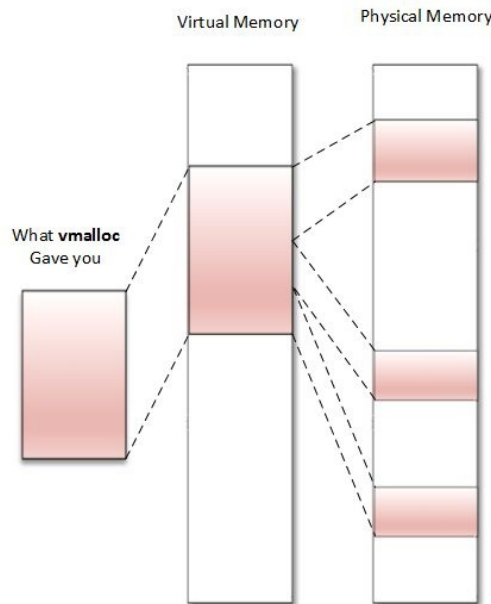
`krealloc()` is the kernel equivalent of the user space `realloc()` function. Because memory returned by `kmalloc()` retains the contents from its previous incarnation, there could be a security risk if it's exposed to user space. To get zeroed `kmalloc`'ed memory, one should use `kzalloc`. `kzfree()` is the freeing function for `kzalloc()`, whereas `kcalloc()` allocates memory for an array, and its parameters `n` and `size` represent respectively the number of elements in the array and the size of an element.



Since `kmalloc()` returns a memory area in the kernel permanent mapping (which mean physically contiguous), the memory address can be translated to a physical address using `virt_to_phys()`, or to a IO bus address using `virt_to_bus()`. These macros internally call either `__pa()` or `__va()` if necessary. The physical address (`virt_to_phys(kmalloc'ed address)`), downshifted by `PAGE_SHIFT`, will produce a PFN of the first page from which the chunk is allocated.

vmalloc allocator

`vmalloc()` is the last kernel allocator we will discuss in the book. It returns memory only contiguous on the virtual space (not physically contiguous):



The returned memory always comes from `HIGH_MEM` zone. Addresses returned cannot be translated into a physical one or into bus address, because one cannot assert that the memory is physically contiguous. It means memory returned by `vmalloc()` can't be used outside the microprocessor (you cannot easily use it for DMA purposes). It is correct to use `vmalloc()` to allocate memory for a large (it does not make sense to use it to allocate one page for example) sequential that exists only in software, for example, a network buffer. It is important to note that `vmalloc()` is slower than `kmalloc()` or page allocator functions, because it must retrieve the memory, build the page tables, or even remap into a virtually contiguous range, whereas `kmalloc()` never does that.

Before using this `vmalloc` API, you should include this header in the code:

```
| #include <linux/vmalloc.h>
```

The following are the `vmalloc` family prototype:

```
void *vmalloc(unsigned long size);
void *vzalloc(unsigned long size);
void vfree( void *addr);
```

`size` is the size of memory you need to allocate. Upon successful allocation of memory, it returns the address of the first byte of the allocated memory block. On failure, it returns a `NULL`. `vfree` function, which is used to free the memory allocated by `vmalloc()`.

The following is an example of using `vmalloc`:

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

void *ptr;
static int alloc_init(void)
{
    unsigned long size = 8192;
    ptr = vmalloc(size);
    if(!ptr)
    {
        /* handle error */
        printk("memory allocation failed\n");
        return -ENOMEM;
    }
    else
        pr_info("Memory allocated successfully\n");
    return 0;
}

static void my_vmalloc_exit(void) /* function called at the time of rmmod */
{
    vfree(ptr); //free the allocated memory
    printk("Memory freed\n");
}
module_init(my_vmalloc_init);
module_exit(my_vmalloc_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("john Madieu, john.madieu@gmail.com");
```

One can use `/proc/vmallocinfo` to display all `vmalloc`'ed memory on the system. `VMALLOC_START` and `VMALLOC_END` are two symbols that delimit the `vmalloc` address range. They are architecture-dependent and defined in `<asm/pgtable.h>`.

Process memory allocation under the hood

Let us focus on the lower level allocator, which allocates pages of memory. The kernel will report allocation of frame pages (physical pages) until really necessary (when those are actually accessed, by reading or writing). This on-demand allocation is called **lazy-allocation**, eliminating the risk of allocating pages that will never be used.

Whenever a page is requested, only the page table is updated, in most of the cases, a new entry is created, which means only virtual memory is allocated. Only when you access the page, an interrupt called **page fault** is raised. This interrupt has a dedicated handler, called the page fault handler, and is called by the MMU in response to an attempt to access virtual memory, which did not immediately succeed.

Actually, a page fault interrupt is raised whatever the access type is (read, write, execute), to a page whose entry in the page table has not got the appropriate permission bits set to allow that type of access. The response to that interrupt falls in one of the following three ways:

- **The hard fault:** The page does not reside anywhere (neither in the physical memory nor a memory-mapped file), which means the handler cannot immediately resolve the fault. The handler will perform I/O operations in order to prepare the physical page needed to resolve the fault, and may suspend the interrupted process and switch to another while the system works to resolve the issue.
- **The soft fault:** The page resides elsewhere in memory (in the working set of another process). It means the fault handler may resolve the fault by immediately attaching a page of physical memory to the appropriate page table entry, adjusting the entry, and resuming the interrupted instruction.
- **The fault cannot be resolved:** This will result in a bus error or segv. `SIGSEGV` is sent to the faulty process, killing it (the default behavior) unless a signal handler has been installed for `SIGSEGV` to change the default behavior.

Memory mappings generally start out with no physical pages attached, by defining the virtual address ranges without any associated physical memory. The actual physical memory is allocated later in response to a page fault exception, when the memory is accessed, since the kernel provides some flags to determine whether the attempted access was legal, and specify the behavior of the page fault handler. Thus, the user space `brk()`, `mmap()` and similar allocate (virtual) space, but physical memory is attached later.



*A page fault occurring in the interrupt context causes a **double fault** interrupt, which usually panics the kernel (calling the `panic()` function) . It is the reason why memory allocated in the interrupt context is taken from a memory pool, which does not raise page fault interrupts. If an interrupt occurs when a double fault is being handled, a triple fault exception is generated, causing the CPU to shut down and the OS immediately reboots. This behavior is actually arc-dependent.*

The copy-on-write (CoW) case

The CoW (heavily used with `fork()`) is a kernel feature that does not allocate several times the memory for a data shared by two or more processes, until a process touches it (write into it); in this case memory is allocated for its private copy. The following shows how a page fault handler manages CoW (one-page case study):

- A PTE is added to the process page table, and marked as un-writable.
- The mapping will result in a VMA creation in the process VMA list. The page is added to that VMA and that VMA is marked as writable.
- On page access (at the first write), the fault handler notices the difference, which means: **this is a Copy on write**. It will then allocate a physical page, which is assigned to the PTE added above, update the PTE flags, flush the TLB entry, and execute the `do_wp_page()` function, which can copy the content from the shared address to the new location.

Work with I/O memory to talk with hardware

Apart from performing data RAM-oriented operations, one can perform I/O memory transactions, to talk with the hardware. When it comes to the access device's register, the kernel offers two possibilities depending on the system architecture:

- **Through the I/O ports:** This is also called **Port Input Output (PIO)**. Registers are accessible through a dedicated bus, and specific instructions (`in` and `out`, in assembler generally) are needed to access those registers. It is the case on x86 architectures.
- **Memory Mapped Input Output (MMIO):** This is the most common and most used method. The device's registers are mapped to memory. Simply read and write to a particular address to write to the registers of the device. It is the case on ARM architectures.

PIO devices access

On a system on which PIO is used, there are two different address spaces, one for memory, which we have already discussed, and the other one for I/O ports, called the port address space, limited to 65,536 ports only. This is a old way, and very uncommon nowadays.

The kernel exports a few functions (symbols) to handle I/O port. Prior to accessing any port regions, we must first inform the kernel that we are using a range of ports using the `request_region()` function, which will return `NULL` on error. Once done with the region, one must call `release_region()`. These are both declared in `linux/ioport.h`. Their prototypes are:

```
struct resource *request_region(unsigned long start,  
                                unsigned long len, char *name);  
void release_region(unsigned long start, unsigned long len);
```

Those functions inform the kernel about your intention to use/release of a region `len` ports, starting from `start`. The `name` parameter should be set with the name of your device. Their use is not mandatory. This is a kind of politeness, which prevents two or more drivers from referencing the same range of ports. One can display information about the ports actually in use on the system by reading the content of `/proc/ioports` files.

Once one is done with region reservation, one can access the port using the following functions:

```
u8 inb(unsigned long addr)  
u16 inw(unsigned long addr)  
u32 inl(unsigned long addr)
```

which respectively access (read) 8-, 16-, or 32-bits sized (wide) ports, and the following functions:

```
void outb(u8 b, unsigned long addr)  
void outw(u16 b, unsigned long addr)  
void outl(u32 b, unsigned long addr)
```

which write `b` data, 8-, 16-, or 32-bits sized, into `addr` port.

The fact that PIO uses a different set of instruction to access I/O ports or MMIO is a disadvantage because PIO requires more instructions than normal memory to accomplish the same task. For instance, 1-bit testing has only one instruction in MMIO, whereas PIO requires reading the data into a register before testing the bit, which is more than one instruction.

MMIO devices access

Memory-mapped I/O reside same address space than memory. The kernel uses part of the address space normally used by RAM (`HIGH_MEM` actually) to map the devices registers, so that instead of having real memory (that is, RAM) at that address, I/O device take place. Thus, communicating to an I/O device becomes like reading and writing to memory addresses devoted to that I/O device.

Like PIO, there are MMIO functions, to inform the kernel about our intention to use a memory region. Remember it is a pure reservation only. These are

`request_mem_region()` and `release_mem_region()`:

```
struct resource* request_mem_region(unsigned long start,  
                                   unsigned long len, char *name)  
void release_mem_region(unsigned long start, unsigned long len)
```

It is also a politeness.



One can display memory regions actually in use on the system by reading the content of the `/proc/iomem` file.

Prior to accessing a memory region (and after you successfully request it), the region must be mapped into kernel address space by calling special architecture-dependent functions (which make use of MMU to build the page table, and thus cannot be called from the interrupt handler). These are `ioremap()` and `iounmap()`, which handle cache coherency too: `void __iomem *ioremap(unsigned long phys_addr, unsigned long size)` `void iounmap(void __iomem *addr)`

`ioremap()` returns a `__iomem void` pointer to the start of the mapped region. Do not be tempted to deference (get/set the value by reading/writing to the pointer) such pointers. The kernel provides functions to access `ioremap`'ed memories. These are: `unsigned int ioread8(void __iomem *addr); unsigned int ioread16(void __iomem *addr); unsigned int ioread32(void __iomem *addr); void iowrite8(u8 value, void __iomem *addr); void iowrite16(u16 value, void __iomem *addr); void iowrite32(u32 value, void __iomem *addr);`



ioremap builds new page tables, just as `vmalloc` does. However, it does not actually allocate any memory but instead, returns a special virtual address that one can use to access the specified physical address range.

On 32-bit systems, the fact that MMIO steals physical memory address space to create mapping for memory-mapped I/O devices is a disadvantage, since it prevents the system from using the stolen memory for general RAM purpose.

__iomem cookie

`__iomem` is a kernel cookie used by Sparse, a semantic checker used by the kernel to find possible coding faults. To take advantage of the features offered by Sparse, it should be enabled at kernel compile time; if not, `__iomem` cookie will be ignored anyway.

The `C=1` in the command line will enable Sparse for you, but parse should be installed first on your system: **sudo apt-get install sparse**

For example, when building a module, use:

```
| make -C $KPATH M=$PWD C=1 modules
```

Alternatively, if the makefile is well written, just type:

```
| make C=1
```

The following shows how `__iomem` is defined in the kernel:

```
| #define __iomem      __attribute__((noderef, address_space(2)))
```

It prevents us from faulty drivers performing I/O memory access. Adding the `__iomem` for all I/O accesses is a way to be stricter too. Since even I/O access is done through virtual memory (on systems with MMU), this cookie prevents us from using absolute physical addresses, and requires us to use `ioremap()`, which will return a virtual address tagged with `__iomem` cookie: `void __iomem *ioremap(phys_addr_t offset, unsigned long size);`

So we can use dedicated functions, such as `ioread23()` and `iowrite32()`. You may wonder why one does not use the `readl()/writel()` function. Those are deprecated, since these do not make sanity checks and are less secure (no `__iomem` required), than `ioreadx()/iowritex()` family functions, which accept only `__iomem` addresses.

In addition, `noderef` is an attribute used by Sparse to make sure programmers do not dereference a `__iomem` pointer. Even though it could work on some architecture, you are not encouraged to do that. Use the special `ioreadx()/iowritex()` function instead. It is portable and works on every

architecture. Now let us see how Sparse will warn us when dereferencing a `__iomem` pointer: `#define BASE_ADDR 0x20E01F8 void *_addrTX = ioremap(BASE_ADDR, 8);`

First, Sparse is not happy because of the wrong type initializer:

```
| warning: incorrect type in initializer (different address spaces)
|   expected void *_addrTX
|   got void [noderef] <asn:2>*
```

Or:

```
| u32 __iomem *_addrTX = ioremap(BASE_ADDR, 8);
| *_addrTX = 0xAABBCCDD; /* bad. No dereference */
| pr_info("%x\n", *_addrTX); /* bad. No dereference */
```

Sparse is still not happy:

```
| Warning: dereference of noderef expression
```

This last example makes Sparse happy:

```
| void __iomem *_addrTX = ioremap(BASE_ADDR, 8);
| iowrite32(0xAABBCCDD, _addrTX);
| pr_info("%x\n", ioread32(_addrTX));
```

The two rules that you must remember are:

- Always use `__iomem` where it is required whether it is as a return type or as a parameter type, and use Sparse to make sure you did so
- Do not dereference a `__iomem` pointer; use a dedicated function instead

Memory (re)mapping

Kernel memory sometimes needs to be remapped, either from kernel to user space, or from kernel to kernel space. The common use case is remapping the kernel memory to user space, but there are other cases, when one needs to access high memory for example.

kmap

Linux kernel permanently maps 896 MB of its address space to the lower 896 MB of the physical memory (low memory). On a 4 GB system, there is only 128 MB left to the kernel to map the remaining 3.2 GB of physical memory (high memory). Low memory is directly addressable by the kernel because of the permanent and one-to-one mapping. When it comes to high memory (memory above 896 MB), the kernel has to map the requested region of high memory into its address space, and the 128 MB mentioned before are especially reserved for this. The function used to perform this trick, `kmap()`, is used to map a given page into the kernel address space.

```
| void *kmap(struct page *page);
```

`page` is a pointer to the `struct page` structure to map. When a high memory page is allocated, it is not directly addressable. `kmap()` is the function one must call to temporarily map high memory into the kernel address space. The mapping will last until `kunmap()` is called:

```
| void kunmap(struct page *page);
```

By temporarily, I mean the mapping should be undone as soon as it is not needed anymore. Remember, 128 MB is not enough to map 3.2 GB. The best programming practice is to unmap high memory mappings when no longer required. It is why the `kmap() - kunmap()` sequence has to be entered around every access to the high memory page. .

This function works on both high memory and low memory. That says, if the page structure resides in low memory, then just the virtual address of the page is returned (because low memory pages already have permanent mappings). If the page belongs to high memory, a permanent mapping is created in the kernel's page tables and the address is returned: `void *kmap(struct page *page) { BUG_ON(in_interrupt()); if (!PageHighMem(page)) return page_address(page); return kmap_high(page); }`

Mapping kernel memory to user space

Mapping physical addresses is one of the most useful functionalities, especially in embedded systems. Sometime you may want to share part of kernel memory with user space. As said earlier, CPU runs in unprivileged mode when running in user space. To let a process access a kernel memory region, we need to remap that region into the process address space.

Using remap_pfn_range

`remap_pfn_range()` maps physical memory (by means of kernel logical address) to a user space process. It is particularly useful for implementing the `mmap()` system call.

After calling the `mmap()` system call on a file (whether it is a device file or not), the CPU will switch to privileged mode, and run the corresponding `file_operations.mmap()` kernel function, which in turn will call `remap_pfn_range()`. The kernel PTE of the mapped region will be derived, and given to the process, of course, with different protection flags. The process's VMA list is updated with a new VMA entry (with appropriate attributes), which will use PTE to access the same memory.

Thus, instead of wasting memory by copying, the kernel just duplicates the PTEs. However, kernel and user space PTE have different attributes.

`remap_pfn_range()` has the following prototype:

```
| int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,  
|                   unsigned long pfn, unsigned long size, pgprot_t flags);
```

A successful call will return 0, and a negative error code on failure. Most of the arguments for `remap_pfn_range()` are provided when the `mmap()` method is called.

- `vma`: This is the virtual memory area provided by the kernel in the case of a `file_operations.mmap()` call. It corresponds to the user process `vma` into which the mapping should be done.
- `addr`: This is the user virtual address where VMA should start (`vma->vm_start`), which will result in a mapping from a virtual address range between `addr` and `addr + size`.
- `pfn`: This represents the page frame number of the kernel memory region to map. It corresponds to the physical address right-shifted by `PAGE_SHIFT` bits. The `vma` offset (offset into the object where the mapping must start) should be taken into account to produce the PFN. Since the `vm_pgoff` field of the VMA structure contains the offset value in the form of the number of pages, it is precisely what you need (with a `PAGE_SHIFT` left-shifting) to extract the

offset in the form of bytes: `offset = vma->vm_pgoff << PAGE_SHIFT`). Finally, `pfn = virt_to_phys(buffer + offset) >> PAGE_SHIFT`.

- **size:** This is the dimension, in bytes, of the area being remapped.
- **prot:** This represents the protection requested for the new VMA. The driver can mangle the default value, but should use the value found in `vma->vm_page_prot` as the skeleton using the OR operator, since some of its bits are already set by user space. Some of these flags are:
 - `VM_IO`, which specifies a device's memory mapped I/O
 - `VM_DONTCOPY`, which tells the kernel not to copy this `vma` on fork
 - `VM_DONTEXPAND`, which prevents `vma` from expanding with `mremap(2)`
 - `VM_DONTDUMP`, prevents the `vma` from being included in the core dump



One may need to modify this value in order to disable caching if using this with I/O memory (`vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);`).

Using `io_remap_pfn_range`

The `remap_pfn_range()` function discussed does not apply anymore when it comes to mapping I/O memory to user space. In this case, the appropriate function is `io_remap_pfn_range()`, whose parameters are the same. The only thing that changes is where the PFN comes from. Its prototype looks like:

```
int io_remap_pfn_range(struct vm_area_struct *vma,
                      unsigned long virt_addr,
                      unsigned long phys_addr,
                      unsigned long size, pgprot_t prot);
```

There is no need to use `ioremap()` when attempting to map I/O memory to user space. `-ioremap()` is intended for kernel purposes (mapping I/O memory into kernel address space), where as `io_remap_pfn_range` is for user space purposes.

Just pass your real physical I/O address (downshifted by `PAGE_SHIFT` to produce a PFN) directly to `io_remap_pfn_range()`. Even if there are some architectures where `io_remap_pfn_range()` is defined as being `remap_pfn_range()`, there are other architectures where it is not the case. For portability reasons, you should only use `remap_pfn_range()` in situations where the PFN parameter points to RAM, and `io_remap_pfn_range()` in situations where `phys_addr` refers to I/O memory.

The mmap file operation

Kernel `mmap` function is part of `struct file_operations` structure, which is executed when the user executes the system call `mmap(2)`, used to map physical memory into a user virtual address. The kernel translates any access to that mapped region of memory through the usual pointer dereferences into a file operation. It is even possible to map device physical memory directly to user space (see `/dev/mem`). Essentially writing to memory becomes like writing into a file. It is just a more convenient way of calling `write()`.

Normally, user space processes cannot access device memory directly for security purposes. Therefore, user space processes use the `mmap()` system call to ask kernel to map the device into the virtual address space of the calling process. After the mapping, the user space process can write directly into the device memory through the returned address.

The `mmap` system call is declared as follows:

```
| mmap (void *addr, size_t len, int prot,  
|       int flags, int fd, off_t offset);
```

The driver should have defined the `mmap` file operation (`file_operations.mmap`) in order to support `mmap(2)`. From the kernel side, the `mmap` field in the driver's file operation structure (`struct file_operations` structure) has the following prototype:

```
| int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

where:

- `filp` is a pointer to the open device file for the driver that results from the translation of the `fd` parameter.
- `vma` is allocated and given as a parameter by the kernel. It is a pointer to the user process's `vma` where the mapping should go. To understand how the kernel creates the new `vma`, let's recall the `mmap(2)` system call's prototype:

```
| void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

The parameters of this function somehow affect some fields of the `vma`:

- `addr`: is the user space's virtual address where the mapping should start. It has an impact on `vma->vm_start`. If `NULL` (the most portable way) was specified, automatically determinate the correct address.
- `length`: This specifies the length of the mapping, and indirectly has an impact on `vma->vm_end`. Remember, the size of a `vma` is always a multiple of `PAGE_SIZE`. In other words, `PAGE_SIZE` is always the smallest size a `vma` can have. The kernel will always alter the size of the `vma` so that it is a multiple of `PAGE_SIZE`.

```

If length <= PAGE_SIZE
    vma->vm_end - vma->vm_start == PAGE_SIZE.
If PAGE_SIZE < length <= (N * PAGE_SIZE)
    vma->vm_end - vma->vm_start == (N * PAGE_SIZE)

```

- `prot`: This affects the permissions of the VMA, which the driver can find in `vma->vm_pro`. As discussed earlier, the driver can update these values, but not alter them.
- `flags`: This determine the type of mapping that the driver can find in `vma->vm_flags`. The mapping can be private or shared.
- `offset`: This specifies the offset within the mapped region, thus mangling the value of `vma->vm_pgoff`.

Implementing mmap in the kernel

Since user space code cannot access kernel memory, the purpose of the `mmap()` function is to derive one or more protected kernel page table entries (which correspond to the memory to be mapped) and duplicate the user space page tables, remove the kernel flag protection, and set permission flags that will allow the user to access the same memory as the kernel without needing special privileges.

The steps to write a `mmap` file operation are as follows:

1. Get the mapping offset and check whether it is beyond our buffer size or not:

```
unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

if (offset >= buffer_size)

    return -EINVAL;
```

2. Check if the mapping size is bigger than our buffer size:

```
unsigned long size = vma->vm_end - vma->vm_start;

if (size > (buffer_size - offset))

    return -EINVAL;
```

3. Get the PFN which corresponds to the PFN of the page where the `offset` position of our buffer falls:


```

unsigned long pfn;

/* we can use page_to_pfn on the struct page structure

* returned by virt_to_page

*/

/* pfn = page_to_pfn (virt_to_page (buffer + offset)); */

/* Or make PAGE_SHIFT bits right-shift on the physical

* address returned by virt_to_phys

*/

pfn = virt_to_phys(buffer + offset) >> PAGE_SHIFT;

```

4. Set the appropriate flag, whether I/O memory is present or not:
 - Disable caching using `vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot)`.
 - Set the `VM_IO` flag: `vma->vm_flags |= VM_IO`.
 - Prevent the VMA from swapping out: `vma->vm_flags |= VM_DONTEXPAND | VM_DONTDUMP`. In kernel versions older than 3.7, you should use only the `VM_RESERVED` flag instead.
5. Call `remap_pfn_range` with the PFN calculated, the size, and the protection flags:

```
if (remap_pfn_range(vma, vma->vm_start, pfn, size, vma->vm_page_prot)) {  
  
    return -EAGAIN;  
  
}  
  
return 0;
```

6. Pass your mmap function to the `struct file_operations` structure:

```
static const struct file_operations my_fops = {  
  
    .owner = THIS_MODULE,  
  
    [...]  
  
    .mmap = my_mmap,  
  
    [...]  
  
};
```

Linux caching system

Caching is the process by which frequently accessed or newly written data is fetched from, or written to a small and faster memory, called a **cache**.

Dirty memory is data-backed (for example, file-backed) memory whose content has been modified (typically in a cache) but not written back to the disk yet. The cached version of the data is newer than the on-disk version, meaning that both versions are out of sync. The mechanism by which cached data is written back on the disk (back store) is called **writeback**. We will eventually update the on-disk version, bringing the two in sync. *Clean memory* is file-backed memory in which the contents are in sync with the disk.

Linux delays write operations in order to speed up the read process, and reduces disk wear leveling by writing data only when necessary. A typical example is the `dd` command. Its complete execution does not mean that the data is written to the target device; this is the reason why `dd` in most cases is chained to a `sync` command.

What is a cache?

A cache is temporary, small, and fast memory used to keep copies of data from larger and often very slow memory, typically placed in systems where there is a working set of data accessed far more often than the rest (for example, hard drive, memory).

When the first read occurs, let us say a process requests some data from the large and slower disk, the requested data is returned to the process, and a copy of accessed data is tracked and cached as well. Any consequent read will fetch data from the cache. Any data modification will be applied in the cache, not on the main disk. Then, the cache region whose content has been modified and differs (is newer than) from the on-disk version will be tagged as **dirty**. When the cache runs full, and since cached data is tracked, new data begins to evict the data that has not been accessed and has been sitting idle for the longest, so that if it is needed again, it will have to be fetched from the large/slow storage again.

CPU cache – memory caching

There are three cache memories on the modern CPU, ordered by size and access speed:

- The **L1** cache that has the smallest amount of memory (often between 1k and 64k) is directly accessible by the CPU in a single clock cycle, which makes it the fastest as well. Frequently used things are in L1 and remain in L1 until some other thing's usage becomes more frequent than the existing one and there is less space in L1. If so, it is moved to a bigger space L2.
- The **L2** cache is the middle level, with a larger amount of memory (up to several megabytes) adjacent to the processor, which can be accessed in a small number of clock cycles. This applies when moving things from L2 to L3.
- The **L3** cache, even slower than L1 and L2, may be two times faster than the main memory (RAM). Each core may have its own L1 and L2 cache; therefore, they all share the L3 cache. Size and speed are the main criteria that change between each cache level: $L1 < L2 < L3$. Whereas original memory access may be 100 ns for example, the L1 cache access can be 0.5 ns.



A real-life example is how a library may put several copies of the most popular titles on display for easy and fast access, but have a large-scale archive with a far greater collection available, at the inconvenience of having to wait for a librarian to go get it for you. The display cases would be analogous to a cache, and the archive would be the large, slow memory.

The main issue that a CPU cache addresses is latency, which indirectly increases the throughput, because access to uncached memory may take a while.

The Linux page cache – disk caching

The page cache, as its name suggests, is a cache of pages in RAM, containing chunks of recently accessed files. The RAM acts as a cache for pages that resides on the disk. In other words, it is the kernel cache of file contents. Cached data may be regular filesystem files, block device files, or memory-mapped files. Whenever a `read()` operation is invoked, the kernel first checks whether the data resides in the page cache, and immediately returns it if found. Otherwise, the data will be read from the disk.



If a process needs to write data without any caching involved, it has to use the `O_SYNC` flag, which guarantees the `write()` command will not return before all data has been transferred to the disk, or the `O_DIRECT` flag, which only guarantees that no caching will be used for data transfer. That says, `O_DIRECT` actually depends on filesystem used and is not recommended.

Specialized caches (user space caching)

- **Web browser cache:** This stores frequently accessed web pages and images onto the disk, instead of fetching them from the web. Whereas the first access to online data may last for more than hundreds of milliseconds, the second access will fetch data from the cache (which is a disk in this case) in only 10 ms.
- **libc or user-app cache:** Memory and disk cache implementations will try to guess what you need to use next, while browser caches keep a local copy in case you need to use it again.

Why delay writing data to disk?

There are essentially two reasons to that:

- Better usage of the disk characteristics; this is efficiency
- Allows the application to continue immediately after a write; this is performance

For example, delaying disk access and processing data only when it reaches a certain size may improve disk performance, and reduce wear leveling of eMMC (on embedded systems). Every chunk write is merged into a single and contiguous write operation. Additionally, written data is cached, allowing the process to return immediately so that any subsequent read will fetch the data from the cache, resulting in a more responsive program. Storage devices prefer a small number of large operations instead of several small operations.

By reporting write operation on the permanent storage later, we can get rid of latency issues introduced by these disks, which are relatively slow.

cat /proc/meminfo | grep Dirty

The flusher threads

The write back cache defers I/O data operations in the page cache. A set of kernel threads, called flusher threads, are responsible for that. Dirty page write back occurs when any one of the following situations is satisfied:

1. When free memory falls below a specified threshold to regain memory consumed by dirty pages.
2. When dirty data lasts until a specific period. The oldest data is written back to the disk to ensure that dirty data does not remain dirty indefinitely.
3. When a user process invokes the `sync()` and `fsync()` system calls. This is an on demand write back.

Device-managed resources – Devres

Devres is a kernel facility helping the developer by automatically freeing the allocated resource in a driver. It simplifies errors handling in `init/probe/open` functions. With devres, each resource allocator has its managed version that will take care of resource release and freeing for you.



This section heavily relies on the Documentation/driver-model/devres.txt file in the kernel source tree, which deals with devres API and lists supported functions along with their descriptions.

The memory allocated with resource-managed functions is associated with the device. devres consists of a linked list of arbitrarily sized memory areas associated with a `struct device`. Each devres resource allocator inserts the allocated resource in the list. The resource remains available until it is manually freed by the code, when the device is detached from the system, or when the driver is unloaded. Each devres entry is associated with a `release` function. There are different ways to release a devres. No matter what, all devres entries are released on driver detach. On release, the associated release function is invoked and then the devres entry is freed.

The following is the list of resources available for a driver:

- Memory for private data structures
- Interrupts (IRQs)
- Memory region allocation (`request_mem_region()`)
- I/O mapping of memory regions (`ioremap()`)
- Buffer memory (possibly with DMA mapping)
- Different framework data structures: Clocks, GPIOs, PWMs, USB phy, regulators, DMA, and so on

Almost every function discussed in this chapter has its managed version. In the majority of cases, the name given to the managed version of a function is obtained by prefixing the original function name with `devm`. For example, `devm_kzalloc()` is the managed version of `kzalloc()`. Additionally, parameters remain

unchanged, but are shifted to the right, since the first parameter is the struct device for which the resource is allocated. There is an exception for functions for which the non-managed version is already given a struct device in its parameters:

```
| void *kmalloc(size_t size, gfp_t flags)  
| void * devm_kmalloc(struct device *dev, size_t size, gfp_t gfp)
```

When the device is detached from the system or the driver for the device is unloaded, that memory is freed automatically. It is possible to free the memory with `devm_kfree()` if it's no longer needed.

The old way:

```
| ret = request_irq(irq, my_isr, 0, my_name, my_data);  
| if(ret) {  
|     dev_err(dev, "Failed to register IRQ.\n");  
|     ret = -ENODEV;  
|     goto failed_register_irq; /* Unroll */  
| }
```

The right way:

```
| ret = devm_request_irq(dev, irq, my_isr, 0, my_name, my_data);  
| if(ret) {  
|     dev_err(dev, "Failed to register IRQ.\n");  
|     return -ENODEV; /* Automatic unroll */  
| }
```

Summary

This chapter is one of the most important chapters. It demystifies memory management and allocation (how and where) in the kernel. Every memory aspect is discussed and detailed, as well as dvres is also explained. The caching mechanism is briefly discussed in order to give an overview of what goes on under the hood during I/O operations. It is a strong base from which introduce and understand the next chapter, which deals with DMA.

DMA – Direct Memory Access

DMA is a feature of computer systems that allows devices to access the main system memory RAM without CPU intervention, which then allows them to devote themselves to other tasks. One typically uses it for accelerating network traffic, but it supports any kind of copy.

The DMA controller is the peripheral responsible for DMA management. One mostly finds it in modern processors and microcontrollers. DMA is a feature used to perform memory read and write operations without stealing CPU cycles. When one needs to transfer a block of data, the processor feeds the DMA controller with the source and destination addresses and the total number of bytes. The DMA controller then transfers the data from the source to the destination automatically, without stealing CPU cycles. When the number of bytes remaining reaches zero, the block transfer ends.

In this chapter, we will cover the following topics:

- Coherent and non-coherent DMA mappings, as well as coherency issues
- DMA engine API
- DMA and DT binding

Setting up DMA mappings

For any type of DMA transfer, one needs to provide source and destination addresses, as well as the number of words to transfer. In the case of a peripheral DMA, the peripheral's FIFO serves as either the source or the destination. When the peripheral serves as the source, a memory location (internal or external) serves as the destination address. When the peripheral serves as the destination, a memory location (internal or external) serves as the source address.

With a peripheral DMA, we specify either the source or the destination, depending on the direction of the transfer. In others words, a DMA transfer requires suitable memory mappings. This is what we will discuss in the following sections.

Cache coherency and DMA

As discussed in [Chapter 11](#), *Kernel Memory Management*, copies of recently accessed memory areas are stored in the cache. This applies to DMA memory too. The reality is that memory shared between two independent devices is generally the source of cache coherency problems. Cache incoherence is an issue coming from the fact that other devices may not be aware of an update from a writing device. On the other hand, cache coherency ensures that every write operation appears to occur instantaneously, so that all devices sharing the same memory region see exactly the same sequence of changes.

A well-explained situation of the coherency issue is illustrated in the following excerpt from LDD3:

Let us imagine a CPU equipped with a cache and an external memory that can be accessed directly by devices using DMA. When the CPU accesses location X in the memory, the current value will be stored in the cache. Subsequent operations on X will update the cached copy of X, but not the external memory version of X, assuming a write-back cache. If the cache is not flushed to the memory before the next time a device tries to access X, the device will receive a stale value of X. Similarly, if the cached copy of X is not invalidated when a device writes a new value to the memory, then the CPU will operate on a stale value of X.

There are actually two ways to address this issue:

- A hardware-based solution. Such systems are **coherent systems**.
- A software-based solution, where the OS is responsible for ensuring cache coherency. One calls such systems **non-coherent systems**.

DMA mappings

Any suitable DMA transfer requires suitable memory mapping. A DMA mapping consists of allocating a DMA buffer and generating a bus address for it. Devices actually use bus addresses. Bus addresses are each instance of the `dma_addr_t` type.

One distinguishes two types of mapping: **coherent DMA mappings** and **streaming DMA mappings**. One can use the former over several transfers, which automatically addresses cache coherency issues. Therefore, it is too expensive. The streaming mapping has a lot of constraints and does not automatically address coherency issues, although, there is a solution for that, which consists of several function calls between each transfer. Coherent mapping usually exists for the life of the driver, whereas one streaming mapping is usually unmapped once the DMA transfer completes.

One should use streaming mapping when one can and coherent mapping when one must.

Back to the code; the main header should include the following to handle DMA mapping: `#include <linux/dma-mapping.h>`

Coherent mapping

The following function sets up a coherent mapping: void

```
*dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle,
gfp_t flag)
```

This function handles both the allocation and the mapping of the buffer, and returns a kernel virtual address for that buffer, which is `size` bytes wide and accessible by the CPU. `dev` is your device structure. The third argument is an output parameter that points to the associated bus address. Memory allocated for the mapping is guaranteed to be physically contiguous, and `flag` determines how memory should be allocated, which is usually `GFP_KERNEL`, OR `GFP_ATOMIC` (if we are in an atomic context).

Do note that this mapping is said to be:

- **Consistent (coherent)**, since it allocates uncached unbuffered memory for a device for performing DMA
- **Synchronous**, because a write by either the device or the CPU can be immediately read by either without worrying about cache coherency

In order to free a mapping, one can use the following function: void

```
dma_free_coherent(struct device *dev, size_t size, void *cpu_addr, dma_addr_t
dma_handle);
```

Here `cpu_addr` corresponds to the kernel virtual address returned by

`dma_alloc_coherent()`. This mapping is expensive, and the minimum it can allocate is a page. In fact, it only allocates the number of pages that is the power of 2.

The order of pages is obtained with `int order = get_order(size)`. One should use this mapping for buffers that last the life of the device.

```
enum dma_data_direction {  
  
    DMA_BIDIRECTIONAL = 0, DMA_TO_DEVICE = 1,  
    DMA_FROM_DEVICE = 2, DMA_NONE = 3, };
```

Single buffer mapping

This is for occasional mapping. One can set up a single buffer with this:

```
dma_addr_t dma_map_single(struct device *dev, void *ptr, size_t size, enum
dma_data_direction direction);
```

The direction should be `DMA_TO_DEVICE`, `DMA_FROM_DEVICE`, OR `DMA_BIDIRECTIONAL`, as described in the preceding code. `ptr` is the kernel virtual address of the buffer, and `dma_addr_t` is the returned bus address for the device. Make sure to use the direction that really fits your need, not just always `DMA_BIDIRECTIONAL`.

One should free the mapping with this:

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr,
size_t size, enum dma_data_direction direction);
```

Scatter/gather mapping

Scatter/gather mappings are a special type of streaming DMA mapping where one can transfer several buffer regions in a single shot, instead of mapping each buffer individually and transferring them one by one. Suppose you have several buffers that might not be physically contiguous, all of which need to be transferred at the same time to or from the device. This situation may occur due to:

- A `readv` or `writv` system call
- A disk I/O request
- Or simply a list of pages in a mapped kernel I/O buffer

The kernel represents the scatterlist as a coherent structure, `struct scatterlist` :

```
struct scatterlist {  
    unsigned long page_link;  
    unsigned int offset;  
    unsigned int length;  
    dma_addr_t dma_address;  
    unsigned int dma_length;  
};
```

In order to set up a scatterlist mapping, one should:

- Allocate your scattered buffers.
- Create an array of the scatter list and fill it with allocated memory using `sg_set_buf()`. Note that scatterlist entries must be of page size (except ends).
- Call `dma_map_sg()` on the scatterlist.
- Once done with DMA, call `dma_unmap_sg()` to unmap the scatterlist.

While one can send contents of several buffers over DMA one at a time by individually mapping each of them, scatter/gather can send them all at once by sending the pointer to the scatterlist to the device, along with a length, which is the number of entries in the list:

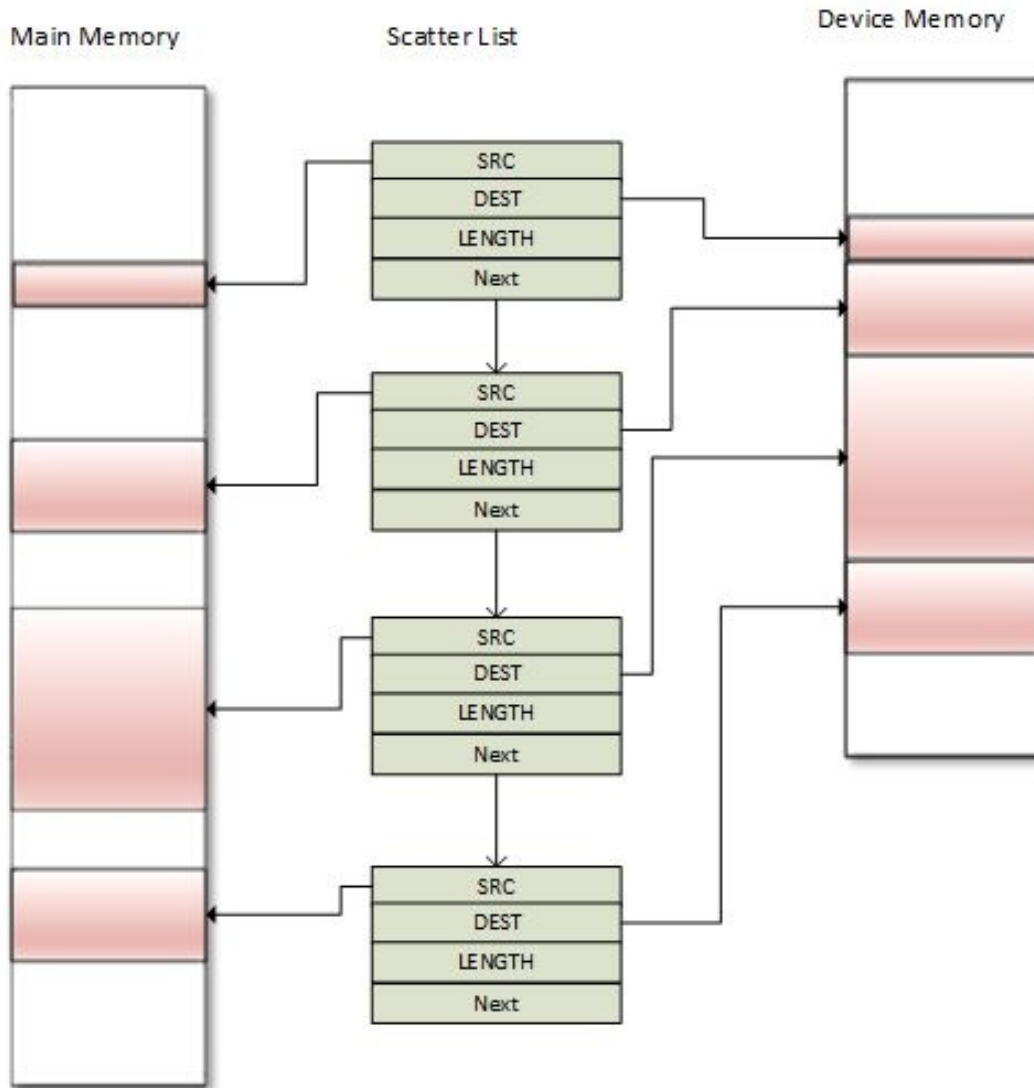
```
u32 *wbuf, *wbuf2, *wbuf3;  
wbuf = kzalloc(SDMA_BUF_SIZE, GFP_DMA);  
wbuf2 = kzalloc(SDMA_BUF_SIZE, GFP_DMA);  
wbuf3 = kzalloc(SDMA_BUF_SIZE/2, GFP_DMA);
```

```

struct scatterlist sg[3];
sg_init_table(sg, 3);
sg_set_buf(&sg[0], wbuf, SDMA_BUF_SIZE);
sg_set_buf(&sg[1], wbuf2, SDMA_BUF_SIZE);
sg_set_buf(&sg[2], wbuf3, SDMA_BUF_SIZE/2);
ret = dma_map_sg(NULL, sg, 3, DMA_MEM_TO_MEM);

```

The same rules described in the single-buffer mapping section apply to scatter/gather.



DMA scatter/gather

`dma_map_sg()` and `dma_unmap_sg()` take care of cache coherency. But if one needs to use the same mapping to access (read/write) the data between the DMA transfer, the buffers must be synced between each transfer in an appropriate manner, by

either `dma_sync_sg_for_cpu()` if the CPU needs to access the buffers, or `dma_sync_sg_for_device()` if it is the device. Similar functions for single region mapping are `dma_sync_single_for_cpu()` and `dma_sync_single_for_device()`:

```
void dma_sync_sg_for_cpu(struct device *dev,
                        struct scatterlist *sg,
                        int nents,
                        enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev,
                           struct scatterlist *sg, int nents,
                           enum dma_data_direction direction);

void dma_sync_single_for_cpu(struct device *dev, dma_addr_t addr,
                             size_t size,
                             enum dma_data_direction dir)

void dma_sync_single_for_device(struct device *dev,
                                dma_addr_t addr, size_t size,
                                enum dma_data_direction dir)
```

There is no need to call the preceding functions again after the buffer(s) has been unmapped. You can just read the content.

Concept of completion

This section will briefly describe completion and the necessary part of its API that the DMA transfer uses. For a complete description, please feel free to have a look at the kernel documentation at *Documentation/scheduler/completion.txt*. A common pattern in kernel programming involves initiating some activity outside of the current thread, then waiting for that activity to complete.

Completion is a good alternative to `sleep()` when waiting for a buffer to be used. It is suitable for sensing data, which is exactly what the DMA callback does.

Working with completion requires this header:

```
| <linux/completion.h>
```

Like other kernel facility data structures, one can create instances of the `struct completion` structure either statically or dynamically:

- Static declaration and initialization looks like this:

```
| DECLARE_COMPLETION(my_comp);
```

- Dynamic allocation looks like this:

```
| struct completion my_comp;  
| init_completion(&my_comp);
```

When the driver begins some work whose completion must be waited for (a DMA transaction in our case), it just has to pass the completion event to the `wait_for_completion()` function:

```
| void wait_for_completion(struct completion *comp);
```

When some other part of the code has decided that the completion has happened (transaction completes), it can wake up anybody (actually the code that needs to access DMA buffer) who is waiting with one of:

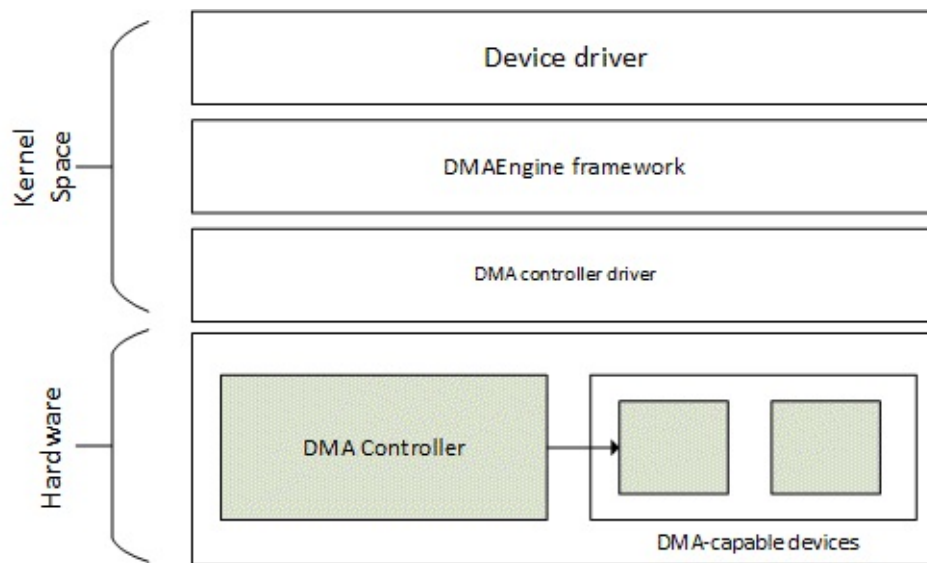
```
| void complete(struct completion *comp);  
| void complete_all(struct completion *comp);
```


As one can guess, `complete()` will wake up only one waiting process, while `complete_all()` will wake up every one waiting for that event. Completions are implemented in such a way that they will work properly even if `complete()` is called before `wait_for_completion()`.

Along with code samples used in the next sections, one will have a better understanding of how this works.

DMA engine API

The DMA engine is a generic kernel framework for developing a DMA controller driver. The main goal of DMA is offloading the CPU when it comes to copy memory. One delegates a transaction (I/O data transfers) to the DMA engine by use of channels. A DMA engine, through its driver/API, exposes a set of channels, which can be used by other devices (slaves).



DMA Engine layout

Here we will simply walk through that (slave) API, which is applicable for slave DMA usage only. The mandatory header here is as follows: `#include <linux/dmaengine.h>`

The slave DMA usage is straightforward, and consists of the following steps:

1. Allocate a DMA slave channel.
2. Set slave and controller specific parameters.
3. Get a descriptor for the transaction.
4. Submit the transaction.
5. Issue pending requests and wait for callback notification.



One can see a DMA channel as a highway for I/O data transfer



Allocate a DMA slave channel

One requests a channel using `dma_request_channel()`. Its prototype is as follows:

```
struct dma_chan *dma_request_channel(const dma_cap_mask_t *mask,  
                                   dma_filter_fn fn, void *fn_param);
```

`mask` is a bitmap mask that represents the capabilities the channel must satisfy. One uses it essentially to specify the transfer types the driver needs to perform:

```
enum dma_transaction_type {  
    DMA_MEMCPY,      /* Memory to memory copy */  
    DMA_XOR,         /* Memory to memory XOR */  
    DMA_PQ,          /* Memory to memory P+Q computation */  
    DMA_XOR_VAL,     /* Memory buffer parity check using XOR */  
    DMA_PQ_VAL,      /* Memory buffer parity check using P+Q */  
    DMA_INTERRUPT,   /* The device is able to generate dummy transfer that will generate */  
    DMA_SG,          /* Memory to memory scatter gather */  
    DMA_PRIVATE,     /* channels are not to be used for global memcpy. Usually used with */  
    DMA_SLAVE,       /* Memory to device transfers */  
    DMA_CYCLIC,      /* Device is able to handle cyclic transfers */  
    DMA_INTERLEAVE,  /* Memory to memory interleaved transfer */  
}
```

The `dma_cap_zero()` and `dma_cap_set()` functions are used to clear the mask and set the capability we need. For example:

```
dma_cap_mask my_dma_cap_mask;  
struct dma_chan *chan;  
dma_cap_zero(my_dma_cap_mask);  
dma_cap_set(DMA_MEMCPY, my_dma_cap_mask); /* Memory to memory copy */  
chan = dma_request_channel(my_dma_cap_mask, NULL, NULL);
```

In the preceding excerpt, `dma_filter_fn` is defined as:

```
typedef bool (*dma_filter_fn)(struct dma_chan *chan,  
                             void *filter_param);
```

If `filter_fn` parameter (which is optional) is `NULL`, `dma_request_channel()` will simply return the first channel that satisfies the capability mask. Otherwise, when the mask parameter is insufficient for specifying the necessary channel, one can use the `filter_fn` routine as a filter for the available channels in the system. The kernel calls the `filter_fn` routine once for each free channel in the system. Upon seeing a suitable channel, `filter_fn` should return `DMA_ACK`, which will tag the given channel to be the return value from `dma_request_channel()`.

A channel allocated through this interface is exclusive to the caller, until `dma_release_channel()` is called:

```
| void dma_release_channel(struct dma_chan *chan)
```

```

int dmaengine_slave_config(struct dma_chan *chan, struct
dma_slave_config *config)

/*

* Please refer to the complete description in *
include/linux/dmaengine.h */

struct dma_slave_config {

    enum dma_transfer_direction direction; phys_addr_t src_addr;
phys_addr_t dst_addr; enum dma_slave_buswidth src_addr_width;
enum dma_slave_buswidth dst_addr_width; u32 src_maxburst; u32
dst_maxburst; [...]

};

struct dma_chan *my_dma_chan; dma_addr_t dma_src, dma_dst;
struct dma_slave_config my_dma_cfg = {0};

/* No filter callback, neither filter param */

my_dma_chan = dma_request_channel(my_dma_cap_mask, 0,
NULL);

/* scr_addr and dst_addr are ignored in this structure for mem to mem
copy */

my_dma_cfg.direction = DMA_MEM_TO_MEM;
my_dma_cfg.dst_addr_width =
DMA_SLAVE_BUSWIDTH_32_BYTES;

dmaengine_slave_config(my_dma_chan, &my_dma_cfg);

```

```
char *rx_data, *tx_data;
```

```
/* No error check */
```

```
rx_data = kzalloc(BUFFER_SIZE, GFP_DMA); tx_data =  
kzalloc(BUFFER_SIZE, GFP_DMA);
```

```
feed_data(tx_data);
```

```
/* get dma addresses */
```

```
dma_src_addr = dma_map_single(NULL, tx_data, BUFFER_SIZE,  
DMA_MEM_TO_MEM); dma_dst_addr = dma_map_single(NULL,  
rx_data, BUFFER_SIZE, DMA_MEM_TO_MEM);
```

In the preceding excerpt, one calls `dma_request_channel()` function in order to take the owner chip of the DMA channel, on which one calls `dmaengine_slave_config()` to apply its configuration.

`dma_map_single()` is called in order to map rx and tx buffers, so that these can be used for purpose of DMA.

```
struct dma_device *dma_dev = my_dma_chan->device; struct  
dma_async_tx_descriptor *tx = NULL;
```

```
tx = dma_dev->device_prep_dma_memcpy(my_dma_chan,  
dma_dst_addr, dma_src_addr, BUFFER_SIZE, 0);
```

```
if (!tx) {
```

```
    printk(KERN_ERR "%s: Failed to prepare DMA transfer\n",  
__FUNCTION__); /* dma_unmap_* the buffer */
```

```
}
```

```
struct dma_async_tx_descriptor *(*device_prep_dma_memcpy)(
```

```
    struct dma_chan *chan, dma_addr_t dst, dma_addr_t src, size_t len,  
    unsigned long flags)
```

```
struct dma_async_tx_descriptor *tx = NULL; tx = dma_dev->  
device_prep_dma_memcpy(my_dma_chan, dma_dst_addr,  
dma_src_addr, BUFFER_SIZE, 0);
```

```
if (!tx) {
```

```
    printk(KERN_ERR "%s: Failed to prepare DMA transfer\n",  
__FUNCTION__); /* dma_unmap_* the buffer */
```

```
}
```

Please have a look at `include/linux/dmaengine.h`, in the definition of a `struct dma_device` structure, to see how all of these hooks are implemented.


```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor
*desc)
```

```
struct completion transfer_ok; init_completion(&transfer_ok); tx-
>callback = my_dma_callback;
```

```
/* Submit our dma transfer */
```

```
dma_cookie_t cookie = dmaengine_submit(tx);
```

```
if (dma_submit_error(cookie)) {
```

```
    printk(KERN_ERR "%s: Failed to start DMA transfer\n",
__FUNCTION__); /* Handle that */
```

```
[...]
```

```
}
```

Issue pending DMA requests and wait for callback notification

Starting the transaction is the last step of the DMA transfer setup. One activates transactions in the pending queue of a channel by calling `dma_async_issue_pending()` on that channel. If the channel is idle then the first transaction in the queue is started and subsequent ones are queued up. On completion of a DMA operation, the next one in the queue is started and a tasklet triggered. This tasklet is in charge of calling the client driver completion callback routine for notification, if set:

```
| void dma_async_issue_pending(struct dma_chan *chan);
```

An example would look like this:

```
| dma_async_issue_pending(my_dma_chan);  
| wait_for_completion(&transfer_ok);  
  
| dma_unmap_single(my_dma_chan->device->dev, dma_src_addr,  
| BUFFER_SIZE, DMA_MEM_TO_MEM);  
| dma_unmap_single(my_dma_chan->device->dev, dma_src_addr,  
|                 BUFFER_SIZE, DMA_MEM_TO_MEM);  
  
| /* Process buffer through rx_data and tx_data virtualaddresses. */
```

The `wait_for_completion()` function will block until our DMA callback gets called, which will update (complete) our completion variable in order to resume the previous blocked code. It is a suitable alternative to `while (!done)`

```
msleep(SOME_TIME);.
```

```
| static void my_dma_callback()  
| {  
|     complete(&transfer_ok);  
|     return;  
| }
```



*The DMA engine API function that actually issues pending transactions is `dmaengine_issue_pending(struct dma_chan *chan)`, which is a wrap around `dma_async_issue_pending()`.*

```
enum sdma_peripheral_type {  
  
    IMX_DMATYPE_SSI, /* MCU domain SSI */  
  
    IMX_DMATYPE_SSI_SP, /* Shared SSI */  
  
    IMX_DMATYPE_MMC, /* MMC */  
  
    IMX_DMATYPE_SDHC, /* SDHC */  
  
    IMX_DMATYPE_UART, /* MCU domain UART */  
  
    IMX_DMATYPE_UART_SP, /* Shared UART */  
  
    IMX_DMATYPE_FIRI, /* FIRI */  
  
    IMX_DMATYPE_CSPI, /* MCU domain CSPI */  
  
    IMX_DMATYPE_CSPI_SP, /* Shared CSPI */  
  
    IMX_DMATYPE_SIM, /* SIM */  
  
    IMX_DMATYPE_ATA, /* ATA */  
  
    IMX_DMATYPE_CCM, /* CCM */  
  
    IMX_DMATYPE_EXT, /* External peripheral */  
  
    IMX_DMATYPE_MSHC, /* Memory Stick Host Controller */  
  
    IMX_DMATYPE_MSHC_SP, /* Shared Memory Stick Host  
Controller */  
  
    IMX_DMATYPE_DSP, /* DSP */  
}
```

```

IMX_DMATYPE_MEMORY, /* Memory */

IMX_DMATYPE_FIFO_MEMORY, /* FIFO type Memory */

IMX_DMATYPE_SPDIF, /* SPDIF */

IMX_DMATYPE_IPU_MEMORY, /* IPU Memory */

IMX_DMATYPE_ASRC, /* ASRC */

IMX_DMATYPE_ESAI, /* ESAI */

IMX_DMATYPE_SSI_DUAL, /* SSI Dual FIFO */

IMX_DMATYPE_ASRC_SP, /* Shared ASRC */

IMX_DMATYPE_SAI, /* SAI */

};

struct imx_dma_data {

    int dma_request; /* DMA request line */

    int dma_request2; /* secondary DMA request line */

    enum sdma_peripheral_type peripheral_type; int priority;

};

enum imx_dma_prio {

    DMA_PRIO_HIGH = 0,

```

```
DMA_PRIO_MEDIUM = 1,  
  
DMA_PRIO_LOW = 2  
  
};  
  
#include <linux/module.h>  
  
#include <linux/slab.h> /* for kmalloc */  
  
#include <linux/init.h>  
  
#include <linux/dma-mapping.h>  
  
#include <linux/fs.h>  
  
#include <linux/version.h>  
  
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(3,0,35))  
#include <linux/platform_data/dma-imx.h> #else  
  
#include <mach/dma.h>  
  
#endif  
  
  
#include <linux/dmaengine.h>  
  
#include <linux/device.h>  
  
  
#include <linux/io.h>  
  
#include <linux/delay.h>
```

```

static int gMajor; /* major number of device */

static struct class *dma_tm_class;

u32 *wbuf; /* source buffer */

u32 *rbuf; /* destinationn buffer */


struct dma_chan *dma_m2m_chan; /* our dma channel */

struct completion dma_m2m_ok; /* completion variable used in the
DMA callback */

#define SDMA_BUF_SIZE 1024

static bool dma_m2m_filter(struct dma_chan *chan, void *param) {

    if (!imx_dma_is_general_purpose(chan)) return false;

    chan->private = param; return true;

}

int sdma_open(struct inode * inode, struct file * filp) {

    dma_cap_mask_t dma_m2m_mask; struct imx_dma_data
m2m_dma_data = {0};

    init_completion(&dma_m2m_ok);

    dma_cap_zero(dma_m2m_mask); dma_cap_set(DMA_MEMCPY,
dma_m2m_mask); /* Set channel capacities */

```

```
    m2m_dma_data.peripheral_type = IMX_DMATYPE_MEMORY; /*  
choose the dma device type. This is proper to i.MX */
```

```
    m2m_dma_data.priority = DMA_PRIO_HIGH; /* we need high  
priority */
```

```
    dma_m2m_chan = dma_request_channel(dma_m2m_mask,  
dma_m2m_filter, &m2m_dma_data); if (!dma_m2m_chan) {
```

```
        printk("Error opening the SDMA memory to memory channel\n");  
return -EINVAL;
```

```
    }
```

```
    wbuf = kzalloc(SDMA_BUF_SIZE, GFP_DMA); if(!wbuf) {
```

```
        printk("error wbuf !!!!!!!!!!!!!\n"); return -1;
```

```
    }
```

```
    rbuf = kzalloc(SDMA_BUF_SIZE, GFP_DMA); if(!rbuf) {
```

```
        printk("error rbuf !!!!!!!!!!!!!\n"); return -1;
```

```
    }
```

```
return 0;
```

```
}
```

```
int sdma_release(struct inode * inode, struct file * filp) {
```

```
    dma_release_channel(dma_m2m_chan); dma_m2m_chan = NULL;
```

```
    kfree(wbuf);
```

```
    kfree(rbuf);
```

```
    return 0;
```

```
}
```

```
ssize_t sdma_read (struct file *filp, char __user * buf, size_t count,  
loff_t * offset)
```

```
{
```

```
    int i;
```

```
    for (i=0; i<SDMA_BUF_SIZE/4; i++) {
```

```
        if (*(rbuf+i) != *(wbuf+i)) {
```

```
            printk("Single DMA buffer copy falled!,r=%x,w=%x,%d\n", *  
(rbuf+i), *(wbuf+i), i); return 0;
```

```
        }
```

```
    }
```

```
    printk("buffer copy passed!\n"); return 0;
```

```
}
```



```

static void dma_m2m_callback(void *data)

{

    printk("in %s\n",__func__); complete(&dma_m2m_ok); return ;

}

ssize_t sdma_write(struct file * filp, const char __user * buf, <br/>
size_t count, loff_t * offset) {

    u32 i;

    struct dma_slave_config dma_m2m_config = {0}; struct
dma_async_tx_descriptor *dma_m2m_desc; /* transaction descriptor
*/

    dma_addr_t dma_src, dma_dst;

    /* No copy_from_user, we just fill the source buffer with predefined
data */

    for (i=0; i<SDMA_BUF_SIZE/4; i++) {

        *(wbuf + i) = 0x56565656; }


    dma_m2m_config.direction = DMA_MEM_TO_MEM;
    dma_m2m_config.dst_addr_width =
DMA_SLAVE_BUSWIDTH_4_BYTES;
    dmaengine_slave_config(dma_m2m_chan, &dma_m2m_config);

    dma_src = dma_map_single(NULL, wbuf, SDMA_BUF_SIZE,
DMA_TO_DEVICE); dma_dst = dma_map_single(NULL, rbuf,

```

```

SDMA_BUF_SIZE, DMA_FROM_DEVICE); dma_m2m_desc =
dma_m2m_chan->device-
>device_prep_dma_memcpy(dma_m2m_chan, dma_dst, dma_src,
SDMA_BUF_SIZE,0); if (!dma_m2m_desc)

    printk("prep error!!\n"); dma_m2m_desc->callback =
dma_m2m_callback; dmaengine_submit(dma_m2m_desc);
dma_async_issue_pending(dma_m2m_chan);
wait_for_completion(&dma_m2m_ok); dma_unmap_single(NULL,
dma_src, SDMA_BUF_SIZE, DMA_TO_DEVICE);
dma_unmap_single(NULL, dma_dst, SDMA_BUF_SIZE,
DMA_FROM_DEVICE);

    return 0;

}

```

```

struct file_operations dma_fops = {

    open: sdma_open,

    release: sdma_release, read: sdma_read,

    write: sdma_write,

};

```

The full code is available in the repository of the book: chapter-12/imx-sdma/imx-sdma-single.c. There is also a module with which to perform the same task, but using scatter/gather mapping: chapter-12/imx-sdma/imx-sdma-scatter-gather.c.

DMA DT binding

DT binding for the DMA channel depends on the DMA controller node, which is SoC dependent, and some parameters (such as DMA cells) may vary from one SoC to another. This example only focuses on the i.MX SDMA controller, which one can find in the kernel source, at *Documentation/devicetree/bindings/dma/fsl-imx-sdma.txt*.

```

uart1: serial@02020000 {

    compatible = "fsl,imx6sx-uart", "fsl,imx21-uart"; reg =
<0x02020000 0x4000>; interrupts = <GIC_SPI 26
IRQ_TYPE_LEVEL_HIGH>; clocks = <&clks
IMX6SX_CLK_UART_IPG>, <&clks
IMX6SX_CLK_UART_SERIAL>; clock-names = "ipg", "per"; dmas
= <&sdma 25 4 0>, <&sdma 26 4 0>; dma-names = "rx", "tx"; status
= "disabled";

};

```

```

static int imx_uart_dma_init(struct imx_port *sport) {

    struct dma_slave_config slave_config = {}; struct device *dev =
sport->port.dev; int ret;

```

```

    /* Prepare for RX : */

```

```

    sport->dma_chan_rx = dma_request_slave_channel(dev, "rx"); if
(!sport->dma_chan_rx) {

```

```

    [...] /* cannot get the DMA channel. handle error */

```

```

    }

```

```

    slave_config.direction = DMA_DEV_TO_MEM;
slave_config.src_addr = sport->port.mapbase + URXD0;
slave_config.src_addr_width =
DMA_SLAVE_BUSWIDTH_1_BYTE; /* one byte less than the

```

watermark level to enable the aging timer */

```
slave_config.src_maxburst = RXTL_DMA - 1; ret =  
dmaengine_slave_config(sport->dma_chan_rx, &slave_config); if  
(ret) {
```

```
    [...] /* handle error */
```

```
}
```

```
    sport->rx_buf = kzalloc(PAGE_SIZE, GFP_KERNEL); if (!sport-  
>rx_buf) {
```

```
    [...] /* handle error */
```

```
}
```

```
/* Prepare for TX : */
```

```
sport->dma_chan_tx = dma_request_slave_channel(dev, "tx"); if  
(!sport->dma_chan_tx) {
```

```
    [...] /* cannot get the DMA channel. handle error */
```

```
}
```

```
    slave_config.direction = DMA_MEM_TO_DEV;  
    slave_config.dst_addr = sport->port.mapbase + URTX0;  
    slave_config.dst_addr_width =
```

```

DMA_SLAVE_BUSWIDTH_1_BYTE; slave_config.dst_maxburst =
TXTL_DMA; ret = dmaengine_slave_config(sport->dma_chan_tx,
&slave_config); if (ret) {

    [...] /* handle error */

}

[...]
```

The magic call here is the `dma_request_slave_channel()` function, which will parse the device node (in the DT) using `of_dma_request_slave_channel()` to gather channel settings, according to the DMA name (refer to the named resource in [Chapter 6, The Concept of Device Tree](#)).

Summary

DMA is a feature that one finds in many modern CPUs. This chapter gives you the necessary steps to get the most out of this device, using the kernel DMA mapping and DMA engine APIs. After this chapter, I have no doubt you will be able to set up at least a memory-to-memory DMA transfer. One can find further information at *Documentation/dmaengine/*, in the kernel source tree. Therefore, the next chapter deals with an entirely different subject—the Linux device model.

Linux Device Model

Until version 2.5, the kernel had no way to describe and manage objects, and the code reusability was not as enhanced as it is now. In other words, there was no device topology nor organization. There was no information on subsystem relationships nor on how the system is put together. Then came the **Linux Device Model (LDM)**, which introduced:

- The concept of class, to group devices of the same type or devices that expose the same functionalities (for example, mice and keyboards are both input devices).
- Communication with the user space through a virtual filesystem called `sysfs`, in order to let user space manage and enumerate devices and the properties they expose.
- Management of object life cycle, using reference counting (heavily used in managed resources).
- Power management in order to handle the order in which devices should shut down.
- The reusability of the code. Classes and frameworks expose interfaces, behaving like contract that any driver that registers with them must respect.
- LDM brought an **Object Oriented (OO)**-like programming style in the kernel.

In this chapter, we will take advantage of LDM and export some properties to the user space through the `sysfs` filesystem.

In this chapter, we will cover the following topics:

- Introducing LDM data structures (Driver, Device, Bus)
- Gathering kernel objects by type
- Dealing with the kernel `sysfs` interface

LDM data structures

The goal is to build a complete DT that will map each physical device present on the system, and introduce their hierarchy. One common and generic structure has been created to represent any object that could be a part of the device model. The upper level of LDM relies on the bus represented in the kernel as an instance of `struct bus_type`; device driver, represented by a `struct device_driver` structure, and device, which is the last element represented as an instance of the `struct device` structure. In this section, we will design a bus driver packt bus, in order to get deep into LDM data structures and mechanisms.

```

struct bus_type {

    const char *name;

    const char *dev_name; struct device *dev_root; struct
device_attribute *dev_attrs; /* use dev_groups instead */

    const struct attribute_group **bus_groups; const struct
attribute_group **dev_groups; const struct attribute_group
**drv_groups;

    int (*match)(struct device *dev, struct device_driver *drv); int
(*probe)(struct device *dev); int (*remove)(struct device *dev); void
(*shutdown)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state); int
(*resume)(struct device *dev);

    const struct dev_pm_ops *pm;

    struct subsys_private *p; struct lock_class_key lock_key; };

#define to_packt_driver(d) container_of(d, struct packt_driver, driver)
#define to_packt_device(d) container_of(d, struct packt_device, dev)

struct packt_device_id {

    char name[PACKT_NAME_SIZE]; kernel_ulong_t driver_data; /*
Data private to the driver */

};

/*

```

* Bus specific device structure

* This is what a packt device structure looks like */

```
struct packt_device {
```

```
    struct module *owner; unsigned char name[30]; unsigned long  
    price; struct device dev;
```

```
};
```

```
/*
```

* Bus specific driver structure

* This is what a packt driver structure looks like * You should provide your device's probe and remove function.

* may be release too

*/

```
struct packt_driver {
```

```
    int (*probe)(struct packt_device *packt); int (*remove)(struct  
    packt_device *packt); void (*shutdown)(struct packt_device *packt);  
};
```

```
/*
```

* Now let us write and export symbols that people writing * drivers for packt devices must use.

*/

```
int packet_register_driver(struct packet_driver *driver) {
```

```
    driver->driver.bus = &packet_bus_type; return  
    driver_register(&driver->driver); }
```

```
EXPORT_SYMBOL(packet_register_driver);
```

```
void packet_unregister_driver(struct packet_driver *driver) {
```

```
    driver_unregister(&driver->driver); }
```

```
EXPORT_SYMBOL(packet_unregister_driver);
```

```
int packet_device_register(struct packet_device *packet) {
```

```
    return device_register(&packet->dev); }
```

```
EXPORT_SYMBOL(packet_device_register);
```

```
void packet_unregister_device(struct packet_device *packet) {
```

```
    device_unregister(&packet->dev); }
```

```
EXPORT_SYMBOL(packet_device_unregister);
```

/*

* This function allocate a bus specific device structure * One must call packt_device_register to register * the device with the bus

*/

```
struct packt_device * packt_device_alloc(const char *name, int id) {
```

```
    struct packt_device *packt_dev; int status;
```

```
    packt_dev = kzalloc(sizeof *packt_dev, GFP_KERNEL); if  
    (!packt_dev)
```

```
        return NULL;
```

```
    /* new devices on the bus are son of the bus device */
```

```
    strcpy(packt_dev->name, name); packt_dev->dev.id = id;  
    dev_dbg(&packt_dev->dev, "device [%s] registered with packt  
    bus\n", packt_dev->name);
```

```
    return packt_dev;
```

```
out_err:
```

```
    dev_err(&adap->dev, "Failed to register packt client %s\n",  
    packt_dev->name); kfree(packt_dev);
```

```
    return NULL;
```

```
}
```

```
EXPORT_SYMBOL_GPL(packt_device_alloc);
```

```
int packt_device_register(struct packt_device *packt) {
```

```
    packt->dev.parent = &packt_bus; packt->dev.bus =  
&packt_bus_type; return device_register(&packt->dev); }
```

```
EXPORT_SYMBOL(packt_device_register);
```

```
/*
```

```
* This is our bus structure
```

```
*/
```

```
struct bus_type packt_bus_type = {
```

```
    .name = "packt", .match = packt_device_match, .probe =  
    packt_device_probe, .remove = packt_device_remove, .shutdown =  
    packt_device_shutdown, };
```

```
/*
```

```
* Bus device, the master.
```

```
*
```

```
*/
```

```
struct device packt_bus = {
```

```
    .release = packt_bus_release, .parent = NULL, /* Root device, no  
    parent needed */
```

```
};
```

```
static int __init packt_init(void) {
```

```
    int status; status = bus_register(&packt_bus_type); if (status < 0)  
    goto err0;
```

```
    status = class_register(&packt_master_class); if (status < 0) goto
```


err1;

/*

* After this call, the new bus device will appear * under /sys/devices in sysfs. Any devices added to this * bus will shows up under /sys/devices/packt-0/.

*/

device_register(&packt_bus);

return 0;

err1:

bus_unregister(&packt_bus_type); err0:

return status; }

int packt_device_register(struct packt_device *packt) {

packt->dev.parent = &packt_bus; packt->dev.bus =
&packt_bus_type; return device_register(&packt->dev); }

EXPORT_SYMBOL(packt_device_register);

```

struct device_driver {

    const char *name; struct bus_type *bus; struct module *owner;

    const struct of_device_id *of_match_table; const struct
acpi_device_id *acpi_match_table;

    int (*probe) (struct device *dev); int (*remove) (struct device *dev);
void (*shutdown) (struct device *dev); int (*suspend) (struct device
*dev, pm_message_t state); int (*resume) (struct device *dev); const
struct attribute_group **groups;

    const struct dev_pm_ops *pm; };

```

```

struct of_device_id {

```

```

    char compatible[128]; const void *data; };

```

- suspend and resume callbacks provide power management functionality. The remove callback is called when the device is physically removed from the system, or when its reference count reaches 0. The remove callback is also called during system reboot.
- probe is the probe callback that runs when attempting to bind a driver to a device. The bus driver is in charge of calling the device driver's probe function.
- group is a pointer to a list (array) of struct attribute_group, used as a default attribute for the driver. Use this method instead of creating an attribute separately.

Device driver registration

`driver_register()` is the low-level function used to register a device driver with the bus. It adds the driver to the bus's list of drivers. When a device driver is registered with the bus, the core walks through the bus's list of devices and calls the bus's match callback for each device that does not have a driver associated with it in order to find out if there are any devices that the driver can handle.

When a match occurs, the device and the device driver are bound together. The process of associating a device with a device driver is called binding.

Now back to the registration of drivers with our packet bus, one has to use `packt_register_driver(struct packt_driver *driver)`, which is a wrapper around `driver_register()`. The `*driver` parameter must have been filled prior to registering the packet driver. The LDM core provides helper functions for iterating over the list of drivers registered with the bus: `int bus_for_each_drv(struct bus_type *bus, struct device_driver * start, void * data, int (*fn)(struct device_driver *, void *))`;

This helper iterates over the bus's list of drivers, and calls the `fn` callback for each driver in the list.

```
struct device {
```

```
    struct device *parent; struct kobject kobj; const struct device_type  
*type; struct bus_type *bus; struct device_driver *driver; void  
*platform_data; void *driver_data;
```

```
    struct device_node *of_node; struct class *class; const struct  
attribute_group **groups; void (*release)(struct device *dev); };
```

- * parent represents the device's parent, used to build device tree hierarchy. When registered with a bus, the bus driver is responsible for setting this field with the bus device.
- * bus represents the bus the device sits on. The bus driver must fill this field.
- * type identifies the device's type.
- kobj is the kobject in handle reference counting and device model support.
- * of_node is a pointer to the OF (DT) node associated with the device. It is up to the bus driver to set this field.
- platform_data is a pointer to the platform data specific to the device. Usually declared in a board-specific file during device provisioning.
- driver_data is a pointer to private data for the driver.
- class is a pointer to the class that the device belongs to.
- * group is a pointer to a list (array) of struct attribute_group, used as the default attribute for the device. Use this method instead of creating the attribute separately.
- release is a callback called when the device reference count reaches zero. The bus has the responsibility of setting this field. The packt bus driver shows you how to do this.

```
int device_add(struct device *dev) {
```

```
    [...]
```

```
    bus_probe_device(dev); if (parent) klist_add_tail(&dev->p-  
>knode_parent, &parent->p->klist_children); [...]
```

```
}
```

```
int bus_for_each_dev(struct bus_type * bus, struct device * start, void  
* data, int (*fn)(struct device *, void *));
```

Whenever a device is added, the core invokes the match method of the bus driver (`bus_type->match`). If the match function says there is a driver for this device, the core will invoke the probe function of the bus driver (`bus_type->probe`), given both device and driver as parameters. It is then up to the bus driver to invoke the probe method of the device's driver (`driver->probe`). For our packt bus driver, the function used to register a device is `packt_device_register(struct packt_device *packt)`, which internally calls `device_register`, and where the parameter is a packt device allocated with `packt_device_alloc`.

Deep inside LDM

The LDM under the wood relies on three important structures, which are `kobject`, `kobj_type`, and `kset`. Let us see how each of these structures are involved in the device model.

```

struct kobject {

    const char *name; struct list_head entry; struct kobject *parent;
    struct kset *kset; struct kobj_type *ktype; struct sysfs_dirent *sd;
    struct kref kref; /* Fields out of our interest have been removed */

};

struct kobject *kobject_create(void) void kobject_init(struct kobject
*kobj, struct kobj_type *ktype)

int kobject_add(struct kobject *kobj, struct kobject *parent, const char
*fmt, ...);

/* Somewhere */

static struct kobject *mykobj;

mykobj = kobject_create();

    if (mykobj) {

        kobject_init(mykobj, &mytype); if (kobject_add(mykobj, NULL,
"%s", "hello")) {

            err = -1; printk("ldm: kobject_add() failed\n");
            kobject_put(mykobj); mykobj = NULL; }

            err = 0; }

static struct kobject * class_kobj = NULL; static struct kobject *
devices_kobj = NULL;

/* Create /sys/class */

```

```
class_kobj = kobject_create_and_add("class", NULL);
```

```
if (!class_kobj) {
```

```
    return -ENOMEM; }
```

```
/* Create /sys/devices */
```

```
devices_kobj = kobject_create_and_add("devices", NULL);
```

```
if (!devices_kobj) {
```

```
    return -ENOMEM; }
```

If a kobject has a `NULL` parent, then `kobject_add` sets parent to `kset`. If both are `NULL`, object becomes a child-member of the top-level `sys` directory


```

struct kobj_type {

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

struct sysfs_ops {

    ssize_t (*show)(struct kobject *kobj, struct attribute *attr, char
    *buf); ssize_t (*store)(struct kobject *kobj, struct attribute *attr, const
    char *buf, size_t size);

};

struct kobj_type *get_ktype(struct kobject *kobj);

static struct sysfs_ops s_ops = {

    .show = show,

    .store = store,

};

static struct kobj_type k_type = {

    .sysfs_ops = &s_ops, .default_attrs = d_attrs, };

static ssize_t show(struct kobject *kobj, struct attribute *attr, char
*buf) {

    struct d_attr *da = container_of(attr, struct d_attr, attr); printk(
"LDM show: called for (%s) attr\n", da->attr.name ); return

```

```
scnprintf(buf, PAGE_SIZE, "%s: %d\n", da->attr.name, da->value); }
```

```
static ssize_t store(struct kobject *kobj, struct attribute *attr, const  
char *buf, size_t len) {
```

```
    struct d_attr *da = container_of(attr, struct d_attr, attr); sscanf(buf,  
"%d", &da->value); printk("LDM store: %s = %d\n", da->attr.name,  
da->value);
```

```
    return sizeof(int);
```

```
}
```

```

struct kset {

    struct list_head list; spinlock_t list_lock; struct kobject kobj; };

struct kset * kset_create_and_add(const char *name, const struct
kset_uevent_ops *u, struct kobject *parent_kobj); void
kset_unregister (struct kset * k);

static struct kobject foo_kobj, bar_kobj;

example_kset = kset_create_and_add("kset_example", NULL,
kernel_kobj); /*

* since we have a kset for this kobject, * we need to set it before
calling the kobject core.

*/

foo_kobj.kset = example_kset;

bar_kobj.kset = example_kset;


retval = kobject_init_and_add(&foo_kobj, &foo_ktype, NULL,
"foo_name"); retval = kobject_init_and_add(&bar_kobj, &bar_ktype,
NULL, "bar_name");

kset_unregister(example_kset);

```

Attribute

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. That said, every data structure that embeds a struct kobject can expose either default attributes provided by the kobject itself (if any), or custom ones. In other words, attributes map kernel data to files in sysfs.

An attribute definition looks like this:

```
struct attribute {  
    char * name;  
    struct module *owner;  
    umode_t mode;  
};
```

The kernel functions used to add/remove attributes from the filesystem are:

```
int sysfs_create_file(struct kobject * kobj,  
                     const struct attribute * attr);  
void sysfs_remove_file(struct kobject * kobj,  
                      const struct attribute * attr);
```

Let us try to define two properties that we will export, each represented by an attribute:

```
struct d_attr {  
    struct attribute attr;  
    int value;  
};  
  
static struct d_attr foo = {  
    .attr.name="foo",  
    .attr.mode = 0644,  
    .value = 0,  
};  
  
static struct d_attr bar = {  
    .attr.name="bar",  
    .attr.mode = 0644,  
    .value = 0,  
};
```

To create each enumerated attribute separately, we have to call the following:

```
sysfs_create_file(mykobj, &foo.attr);  
sysfs_create_file(mykobj, &bar.attr);
```

A good place to start with attributes is `samples/kobject/kobject-example.c` in the kernel source.

Attributes group

So far, we have seen how to individually add attributes and call (directly or indirectly through a wrapper function such as `device_create_file()`, `class_create_file()`, and so on) `sysfs_create_file()` on each of them. Why bother ourselves with multiple calls if we can do it once? Here is where the attribute group comes in. It relies on the `struct attribute_group` structure: `struct attribute_group { struct attribute **attrs; };`

Of course, we have removed fields that are not of interest. The `attrs` field is a pointer to `NULL` terminated list of attributes. Each attribute group must be given a pointer to a list/array of `struct attribute` elements. The group is just a helper wrapper that makes it easier to manage multiple attributes.

The kernel functions used to add/remove group attributes to the filesystem are:

```
| int sysfs_create_group(struct kobject *kobj,  
|                       const struct attribute_group *grp)  
| void sysfs_remove_group(struct kobject * kobj,  
|                         const struct attribute_group * grp)
```

The two preceding defined properties can be embedded in a `struct attribute_group`, to make only one call to add both of them to the system: `static struct d_attr foo = { .attr.name="foo", .attr.mode = 0644, .value = 0, }; static struct d_attr bar = { .attr.name="bar", .attr.mode = 0644, .value = 0, }; /* attrs is a pointer to a list (array) of attributes */ static struct attribute * attrs [] = { &foo.attr, &bar.attr, NULL, }; static struct attribute_group my_attr_group = { .attrs = attrs, };`

The one and only function to call here is this:

```
| sysfs_create_group(mykobj, &my_attr_group);
```

It is much better than making a call for each attribute.

Device model and sysfs

`sysfs` is a non-persistent virtual filesystem that provides a global view of the system and exposes the kernel object's hierarchy (topology) by means of their kobjects. Each kobject shows up as a directory, and files in a directory representing kernel variables, exported by the related kobject. These files are called attributes, and can be read or written.

If any registered kobject creates a directory in `sysfs`, where the directory is created depends on the kobject's parent (which is a kobject too). It is natural that directories are created as subdirectories of the kobject's parent. This highlights internal object hierarchies to the user space. Top-level directories in `sysfs` represent the common ancestors of object hierarchies, that is, the subsystems the objects belong to.

Top-level `sysfs` directories can be found under the `/sys/` directory:

```
/sys$ tree -L 1
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power
```

`block` contains a directory per-block device on the system, each of which contains subdirectories for partitions on the device. `bus` contains the registered bus on the system. `dev` contains the registered device nodes in a raw way (no hierarchy), each being a symlink to the real device in the `/sys/devices` directory. `devices` gives a view of the topology of devices in the system. `firmware` shows a system-specific tree of low-level subsystems, such as: ACPI, EFI, OF (DT). `fs` lists filesystems actually used on the system. `kernel` holds kernel configuration options and status info. `modules` is a list of loaded modules.

Each of these directories corresponds to a kobject, some of which are exported

as kernel symbols. These are:

- `kernel_kobj` which corresponds to `/sys/kernel`
- `power_kobj` for `/sys/power`
- `firmware_kobj` which is for `/sys/firmware`, exported in the `drivers/base/firmware.c` source file
- `hypervisor_kobj` for `/sys/hypervisor`, exported in the `drivers/base/hypervisor.c`
- `fs_kobj` which corresponds to `/sys/fs`, exported in the `fs/namespace.c` file

However, `class/`, `dev/`, `devices/`, are created during the boot by the `devices_init` function in `drivers/base/core.c` in kernel source, `block/` is created in `block/genhd.c`, and `bus/` is created as a `kset` in `drivers/base/bus.c`.

When a `kobject` directory is added to `sysfs` (using `kobject_add`), where it is added depends on the `kobject`'s parent location. If its parent pointer is set, it is added as a subdirectory inside the parent's directory. If the parent pointer is `NULL`, it is added as a subdirectory inside `kset->kobj`. If neither parent nor `kset` fields are set, it maps to the root level directory in `sysfs` (`/sys`).

One can create/remove symbolic links on existing objects (directories), using `sysfs_{create|remove}_link` functions:

```
int sysfs_create_link(struct kobject * kobj,
                     struct kobject * target, char * name);
void sysfs_remove_link(struct kobject * kobj, char * name);
```

This will allow an object to exist in more than one place. The create function will create a symlink named `name` pointing to the `target` `kobject` `sysfs` entry. A well know example is `devices` appearing in both `/sys/bus` and `/sys/devices`. Symbolic links created will be persistent even after `target` removal. You have to know when the `target` is removed, and then remove the corresponding symlink.


```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);  
void sysfs_remove_file(struct kobject *kobj, const struct attribute  
*attr); int sysfs_create_group(struct kobject *kobj, const struct  
attribute_group *grp); void sysfs_remove_group(struct kobject * kobj,  
const struct attribute_group * grp);
```

Current interfaces

There are interface layers that currently exist in sysfs. Apart from creating your own ktype or kobject to add your attributes, you can use ones that currently exist: device, driver, bus, and class attributes. Their description are as follows:

Device attributes

Apart from default attributes provided by the kobject embedded in your device structure, you can create custom ones. The structure used for this purpose is `struct device_attribute`, which is nothing but a wrapping around the standard `struct attribute`, and a set of callbacks to show/store the value of the attribute:

```
struct device_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device *dev,
                    struct device_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct device *dev,
                    struct device_attribute *attr,
                    const char *buf, size_t count);
};
```

Their declaration is done through the `DEVICE_ATTR` macro:

```
| DEVICE_ATTR(_name, _mode, _show, _store);
```

Whenever you declare a device attribute using `DEVICE_ATTR`, the prefix `dev_attr_` is added to the attribute name. For example, if you declare an attribute with the `_name` parameter set to `foo`, the attribute will be accessible through the `dev_attr_foo` variable name.

To understand why, let us see how the `DEVICE_ATTR` macro is defined in `include/linux/device.h`:

```
| #define DEVICE_ATTR(_name, _mode, _show, _store) \
|     struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

Finally, you can add/remove those using the `device_create_file` and `device_remove_file` functions:

```
| int device_create_file(struct device *dev,
|                       const struct device_attribute * attr);
| void device_remove_file(struct device *dev,
|                       const struct device_attribute * attr);
```

The following sample is a demonstration of how to put it all together:

```
| static ssize_t foo_show(struct device *child,
|                       struct device_attribute *attr, char *buf)
```

```

{
    return sprintf(buf, "%d\n", foo_value);
}

static ssize_t bar_show(struct device *child,
                        struct device_attribute *attr, char *buf)
{
    return sprintf(buf, "%d\n", bar_value);
}

```

Here are the static declarations of the attribute:

```

static DEVICE_ATTR(foo, 0644, foo_show, NULL);
static DEVICE_ATTR(bar, 0644, bar_show, NULL);

```

The following code shows how to actually create files on the system:

```

if ( device_create_file(dev, &dev_attr_foo) != 0 )
    /* handle error */

if ( device_create_file(dev, &dev_attr_bar) != 0 )
    /* handle error*/

```

For cleanup, the attribute removal is done in the remove function as follows:

```

device_remove_file(wm->dev, &dev_attr_foo);
device_remove_file(wm->dev, &dev_attr_bar);

```

You may wonder how and why we used to define the same set of store/show callbacks for all attributes of the same kobject/ktype, and now, we use a custom one for each attribute. The first reason is because, the device subsystem defines its own attribute structure, which wraps the standard one, and secondly, instead of showing/storing the value of the attribute, it uses the `container_of` macro to extract the `struct device_attribute` giving a generic `struct attribute`, and then executes the show/store callback depending on the user action. The following is the excerpt from `drivers/base/core.c`, showing `sysfs_ops` of the device kobject:

```

static ssize_t dev_attr_show(struct kobject *kobj,
                             struct attribute *attr,
                             char *buf)
{
    struct device_attribute *dev_attr = to_dev_attr(attr);
    struct device *dev = kobj_to_dev(kobj);
    ssize_t ret = -EIO;

    if (dev_attr->show)
        ret = dev_attr->show(dev, dev_attr, buf);
    if (ret >= (ssize_t)PAGE_SIZE) {
        print_symbol("dev_attr_show: %s returned bad count\n",
                     (unsigned long)dev_attr->show);
    }
    return ret;
}

```

```

static ssize_t dev_attr_store(struct kobject *kobj, struct attribute *attr,
                             const char *buf, size_t count)
{
    struct device_attribute *dev_attr = to_dev_attr(attr);
    struct device *dev = kobj_to_dev(kobj);
    ssize_t ret = -EIO;

    if (dev_attr->store)
        ret = dev_attr->store(dev, dev_attr, buf, count);
    return ret;
}

static const struct sysfs_ops dev_sysfs_ops = {
    .show = dev_attr_show,
    .store = dev_attr_store,
};

```

The principle is the same for bus (in `drivers/base/bus.c`), driver (in `drivers/base/bus.c`), and class (in `drivers/base/class.c`) attributes. They use the `container_of` macro to extract their specific attribute structure, and then call the `show/store` callback embedded in it.

Bus attributes

It relies on the `struct bus_attribute` structure: `struct bus_attribute { struct attribute attr; ssize_t (*show)(struct bus_type *, char * buf); ssize_t (*store)(struct bus_type *, const char * buf, size_t count); };`

Bus attributes are declared using the `BUS_ATTR` macro: `BUS_ATTR(_name, _mode, _show, _store)`

Any bus attribute declared using `BUS_ATTR` will have the prefix `bus_attr_` added to the attribute variable name: `#define BUS_ATTR(_name, _mode, _show, _store) \ struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show, _store)`

They are created/removed using `bus_{create|remove}_file` functions: `int bus_create_file(struct bus_type *, struct bus_attribute *); void bus_remove_file(struct bus_type *, struct bus_attribute *);`

Device drivers attributes

The structure used is `struct driver_attribute`: `struct driver_attribute { struct attribute attr; ssize_t (*show)(struct device_driver *, char * buf); ssize_t (*store)(struct device_driver *, const char * buf, size_t count); };`

The declaration relies on the `DRIVER_ATTR` macro, which will prefix the attribute variable name with `driver_attr_`: `DRIVER_ATTR(_name, _mode, _show, _store)`

The macro definition is:

```
#define DRIVER_ATTR(_name, _mode, _show, _store) \
struct driver_attribute driver_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

Creation/removal relies on `driver_{create|remove}_file` functions: `int driver_create_file(struct device_driver *, const struct driver_attribute *); void driver_remove_file(struct device_driver *, const struct driver_attribute *);`

```

struct class_attribute {

    struct attribute attr; ssize_t (*show)(struct device_driver *, char *
buf); ssize_t (*store)(struct device_driver *, const char * buf, size_t
count);

};

CLASS_ATTR(_name, _mode, _show, _store)

#define CLASS_ATTR(_name, _mode, _show, _store) \

struct class_attribute class_attr_##_name = __ATTR(_name, _mode,
_show, _store)

int class_create_file(struct class *class, const struct class_attribute
*attr);

void class_remove_file(struct class *class, const struct class_attribute
*attr);

int device_create_file(struct device *dev, const struct device_attribute
*attr) {

    [...]

    error = sysfs_create_file(&dev->kobj, &attr->attr); [...]

}

int class_create_file(struct class *cls, const struct class_attribute *attr)
{

```



```
[...]
```

```
error =
```

```
sysfs_create_file(&cls->p->class_subsys.kobj, &attr->attr); return  
error;
```

```
}
```

```
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr) {
```

```
[...]
```

```
error =
```

```
sysfs_create_file(&bus->p->subsys.kobj, &attr->attr); [...]
```

```
}
```

Allow sysfs attribute files to be pollable

Here we will see how not to make CPU wasting polling to sense sysfs attributes data availability. The idea is to use the `poll` or `select` system calls to wait for the attribute's content to change. The patch to make sysfs attributes pollable was created by **Neil Brown** and **Greg Kroah-Hartman**. The kobject manager (the driver which has access to the kobject) must support notification to allow `poll` or `select` to return (be released) when the content changes. The magic function that does the trick comes from the kernel side, and is `sysfs_notify()`:

```
void sysfs_notify(struct kobject *kobj, const char *dir,
                  const char *attr)
```

If the `dir` parameter is non-NULL, it is used to find a subdirectory, which contains the attribute (presumably created by `sysfs_create_group`). This has a cost of one `int` per attribute, one `wait_queuehead` per kobject, one `int` per open file.

`poll` will return `POLLERR|POLLPRI`, and `select` will return the fd whether it is waiting for read, write, or exceptions. The blocking poll is from the user's side.

`sysfs_notify()` should be called only after you adjust your kernel attribute value.



*Think of the `poll()` (or `select()`) code as a **subscriber** to notice a change in an attribute of interest, and `sysfs_notify()` as a **publisher**, notifying subscribers of any changes.*

The following is an excerpt of code provided with the book, which is the store function of an attribute:

```
static ssize_t store(struct kobject *kobj, struct attribute *attr,
                    const char *buf, size_t len)
{
    struct d_attr *da = container_of(attr, struct d_attr, attr);

    sscanf(buf, "%d", &da->value);
    printk("sysfs_foo store %s = %d\n", a->attr.name, a->value);

    if (strcmp(a->attr.name, "foo") == 0){
        foo.value = a->value;
        sysfs_notify(mykobj, NULL, "foo");
    }
}
```

```
    else if(strcmp(a->attr.name, "bar") == 0){  
        bar.value = a->value;  
        sysfs_notify(mykobj, NULL, "bar");  
    }  
    return sizeof(int);  
}
```

The code from the user space must behave like this in order to sense the data change:

1. Open the file attributes.
2. Make a dummy read of all the contents.
3. Call poll requesting `POLLERR|POLLPRI` (select/excceptfds works too).
4. When `poll` (or `select`) returns (which indicates that a value has changed), read the content of files whose data changed.
5. Close the files and go to the top of the loop.

When in doubt of a sysfs attribute being pollable, set a suitable timeout value. The user space example is provided with the book sample.

Summary

Now you are familiar with the LDM concept and with its data structures (bus, class, device drivers, and devices), including low-level data structures, which are `kobject`, `kset`, `kobj_types`, and attributes (or group of those), how objects are represented within the kernel (hence `sysfs` and devices topology) is not a secret anymore. You will be able to create an attribute (or group), exposing your device or driver feature through `sysfs`. If the previous topic seems clear to you, we will move to the next [chapter 14](#), *Pin Control and GPIO Subsystem*, which heavily uses the power of `sysfs`.

Pin Control and GPIO Subsystem

Most embedded Linux driver and kernel engineers write using GPIOs or play with pins multiplexing. By pins, I mean outgoing line of component. SoC does multiplex pins, meaning that a pin may have several functions, for example, `MX6QDL_PAD_SD3_DAT1` in `arch/arm/boot/dts/imx6dl-pinctrl.h` can be either an SD3 data line 1, UART1's cts/rts, Flexcan2's Rx, or normal GPIO.

The mechanism by which one chooses the mode a pin should work on is called pin muxing. The system responsible for is called the pin controller. In the second part of the chapter, we will discuss the **General Purpose Input Output (GPIO)**, which is a special function (mode) in which a pin can operate.

In this chapter, we will:

- Walk through the pin control subsystem, and see how one can declare their nodes in DT
- Explore both legacy integer-based GPIO interfaces, as well as the new descriptor-based interface API
- Deal with GPIO mapped to IRQ
- Handle sysfs interfaces dedicated to GPIOs

Pin control subsystem

The **Pin control (pinctrl)** subsystem allows managing pin muxing. In the DT, devices that need pins to be multiplexed in a certain way must declare the pin control configuration they need.

The pinctrl subsystem provides:

- Pin multiplexing, which allows for reusing the same pin for different purposes, such as one pin being a UART TX pin, GPIO line, or HSI data line. Multiplexing can affect groups of pins or individual pins.
- Pin configuration, applying electronic properties of pins such as pull-up, pull-down, driver strength, debounce period, and so on.

The purpose of this book is limited to using functions exported by the pin controller driver, and does not not how to write a pin controller driver.

Pinctrl and the device tree

The pinctrl is nothing but a way to gather pins (not only GPIO), and pass them to the driver. The pin controller driver is responsible for parsing pin descriptions in the DT and applying their configuration in the chip. The driver usually needs a set of two nested nodes to describe group of pins configurations. The first node describes the function of the group (what purpose the group will be used for), the second holds the pins configuration.

How pin groups are assigned in the DT heavily depends on the platform, and thus the pin controller driver. Every pin control state is given an integer ID starting at 0 and contiguous. One can use a name property, which will be mapped on top of IDs, so that the same name always points to the same ID.

Each client device's own binding determines the set of states that must be defined in its DT node, and whether to define the set of state IDs that must be provided, or whether to define the set of state names that must be provided. In any case, a pin configuration node can be assigned to a device by means of two properties:

- `pinctrl-<ID>`: This allows for giving 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 configuration nodes must be child nodes of the pin controller that they configure. Multiple entries may exist in this list so that multiple pin controllers may be configured, or so that a state may be built from multiple nodes for a single pin controller, each contributing part of the overall configuration.
- `pinctrl-name`: This allows for giving a name to each state in a list. List entry 0 defines the name for integer state ID 0, list entry 1 for state ID 1, and so on. The state ID 0 is commonly given the name *default*. The list of standardized states can be found in `include/linux/pinctrl/pinctrl-state.h`.
- The following is an excerpt of DT, showing some device nodes, along with their pin control nodes:

```
usdhc@0219c000 { /* uSDHC4 */
    non-removable;
    vmmc-supply = <&reg_3p3v>;
```

```

    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_usdhc4_1>;
};

gpio-keys {
    compatible = "gpio-keys";
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_io_foo &pinctrl_io_bar>;
};

iomuxc@020e0000 {
    compatible = "fsl,imx6q-iomuxc";
    reg = <0x020e0000 0x4000>;

    /* shared pinctrl settings */
    usdhc4 { /* first node describing the function */
        pinctrl_usdhc4_1: usdhc4grp-1 { /* second node */
            fsl,pins = <
                MX6QDL_PAD_SD4_CMD__SD4_CMD      0x17059
                MX6QDL_PAD_SD4_CLK__SD4_CLK       0x10059
                MX6QDL_PAD_SD4_DAT0__SD4_DATA0     0x17059
                MX6QDL_PAD_SD4_DAT1__SD4_DATA1     0x17059
                MX6QDL_PAD_SD4_DAT2__SD4_DATA2     0x17059
                MX6QDL_PAD_SD4_DAT3__SD4_DATA3     0x17059
                MX6QDL_PAD_SD4_DAT4__SD4_DATA4     0x17059
                MX6QDL_PAD_SD4_DAT5__SD4_DATA5     0x17059
                MX6QDL_PAD_SD4_DAT6__SD4_DATA6     0x17059
                MX6QDL_PAD_SD4_DAT7__SD4_DATA7     0x17059
            >;
        };
    };
    [...]
    uart3 {
        pinctrl_uart3_1: uart3grp-1 {
            fsl,pins = <
                MX6QDL_PAD_EIM_D24__UART3_TX_DATA 0x1b0b1
                MX6QDL_PAD_EIM_D25__UART3_RX_DATA  0x1b0b1
            >;
        };
    };
    // GPIOs (Inputs)
    gpios {
        pinctrl_io_foo: pinctrl_io_foo {
            fsl,pins = <
                MX6QDL_PAD_DISP0_DAT15__GPIO5_I009 0x1f059
                MX6QDL_PAD_DISP0_DAT13__GPIO5_I007 0x1f059
            >;
        };
        pinctrl_io_bar: pinctrl_io_bar {
            fsl,pins = <
                MX6QDL_PAD_DISP0_DAT11__GPIO5_I005 0x1f059
                MX6QDL_PAD_DISP0_DAT9__GPIO4_I030  0x1f059
                MX6QDL_PAD_DISP0_DAT7__GPIO4_I028  0x1f059
                MX6QDL_PAD_DISP0_DAT5__GPIO4_I026  0x1f059
            >;
        };
    };
};

```

In the preceding example, a pin configuration is given in the form <PIN_FUNCTION> <PIN_SETTING>. For example:


```
| MX6QDL_PAD_DISP0_DAT15__GPIO5_I009 0x80000000
```

MX6QDL_PAD_DISP0_DAT15__GPIO5_I009 represents the pin function, which is GPIO in this case, and 0x80000000 represents the pin settings.

For this line,

```
| MX6QDL_PAD_EIM_D25__UART3_RX_DATA 0x1b0b1
```

MX6QDL_PAD_EIM_D25__UART3_RX_DATA represents the pin function, which is the RX line of UART3, and 0x1b0b1 represent is settings.

The pin function is a macro whose value is meaningful for pin controller driver only. These are generally defined in header files located in `arch/<arch>/boot/dts/`. If one uses a UDOO quad, for example, which has an i.MX6 quad core (ARM), the pin function header would be `arch/arm/boot/dts/imx6q-pinctrl.h`. The following is the macro corresponding to the fifth line of the GPIO5 controller:

```
| #define MX6QDL_PAD_DISP0_DAT11__GPIO5_I005 0x19c 0x4b0 0x000 0x5 0x0
```

<PIN_SETTING> can be used to set up things like pull-ups, pull-downs, keepers, drive strength, and so on. How it should be specified depends on the pin controller binding, and the meaning of its value depends on the SoC data sheet, generally in the IOMUX section. On i.MX6 IOMUXC, only lower than 17 bits are used for this purpose.

These preceding nodes are called from the corresponding driver-specific node. Moreover, these pins are configured during corresponding driver initialization. Prior to selecting a pin group state, one must get the pin control first using the `pinctrl_get()` function, call `pinctrl_lookup_state()` in order to check whether the requested state exist or not, and finally `pinctrl_select_state()` to apply the state.

The following is a sample that shows how to get a pincontrol and apply its default configuration:

```
struct pinctrl *p;
struct pinctrl_state *s;
int ret;

p = pinctrl_get(dev);
if (IS_ERR(p))
    return p;

s = pinctrl_lookup_state(p, name);
```

```

if (IS_ERR(s)) {
    devm_pinctrl_put(p);
    return ERR_PTR(PTR_ERR(s));
}

ret = pinctrl_select_state(p, s);
if (ret < 0) {
    devm_pinctrl_put(p);
    return ERR_PTR(ret);
}

```

One usually performs such steps during driver initialization. The suitable place for this code could be within the `probe()` function.



`pinctrl_select_state()` internally calls `pinmux_enable_setting()`, which in turn calls the `pin_request()` on each pin in the pin control node.

A pin control can be released with the `pinctrl_put()` function. One can use the resource-managed version of the API. That said, one can use `pinctrl_get_select()`, given the name of the state to select, in order to configure pinmux. The function is defined in `include/linux/pinctrl/consumer.h` as follows:

```

static struct pinctrl *pinctrl_get_select(struct device *dev,
                                          const char *name)

```

where `*name` is the state name as written in `pinctrl-name` property. If the name of the state is default, one can just call `pinctrl_get_select_default()` function, which is a wrapper around `pinctrl_get_select()`:

```

static struct pinctrl * pinctrl_get_select_default(
    struct device *dev)
{
    return pinctrl_get_select(dev, PINCTRL_STATE_DEFAULT);
}

```

Let us see a real example in a board-specific dts file (`am335x-evm.dts`):

```

dcan1: d_can@481d0000 {
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&d_can1_pins>;
};

```

And in the corresponding driver:

```

pinctrl = devm_pinctrl_get_select_default(&pdev->dev);
if (IS_ERR(pinctrl))
    dev_warn(&pdev->dev, "pins are not configured from the driver\n");

```



The pin control core will automatically claim the `default` `pinctrl` state for us when the device is probed. If one defines an `init` state, the `pinctrl` core will automatically set `pinctrl` to this state before the `probe()` function, and then switch to the `default` state after `probe()` (unless the driver explicitly changed states already).

The GPIO subsystem

From the hardware point of view, a GPIO is a functionality, a mode in which a pin can operate. From a software point of view, a GPIO is nothing but a digital line, which can operate as an input or output, and can have only two values: (1 for high or 0 for low). Kernel GPIO subsystems provide every function you can imagine to set up and handle GPIO line from within your driver:

- Prior to using a GPIO from within the driver, one should claim it to the kernel. This is a way to take the ownership of the GPIO, preventing other drivers from accessing the same GPIO. After taking the ownership of the GPIO, one can:
 - Set the direction
 - Toggle its output state (driving line high or low) if used as output
 - Set the debounce-interval and read the state, if used as input. For GPIO lines mapped to IRQ, one can define at what edge/level the interrupt should be triggered, and register a handler that will be run whenever the interrupt occurs.

There are actually two different ways to deal with GPIO in the kernel, as follows:

- The legacy and depreciated integer-based interface, where GPIOs are represented by integer
- The new and recommended descriptor-based interface, where a GPIO is represented and described by an opaque structure, with a dedicated API

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

There are well known functions to handle GPIO in kernel.

Claiming and configuring the GPIO

One can allocate and take the ownership of a GPIO using the `gpio_request()` function:

```
| static int gpio_request(unsigned gpio, const char *label)
```

`gpio` represents the GPIO number we are interested in, and `label` is the label used by the kernel for the GPIO in sysfs, as we can see in `/sys/kernel/debug/gpio`. You have to check the value returned, where `0` mean success, and negative error code on error. Once done with the GPIO, it should be set free with the `gpio_free()` function:

```
| void gpio_free(unsigned int gpio)
```

If in doubt, one can use `gpio_is_valid()` function to check whether this GPIO number is valid on the system prior to allocate it:

```
| static bool gpio_is_valid(int number)
```

Once we own the GPIO, we can change its direction, depending on the need, and whether it should be an input or output, using the `gpio_direction_input()` OR `gpio_direction_output()` functions:

```
| static int gpio_direction_input(unsigned gpio)
| static int gpio_direction_output(unsigned gpio, int value)
```

`gpio` is the GPIO number we need to set the direction. There is a second parameter when it comes to configuring the GPIO as output: `value`, which is the state the GPIO should be in once the output direction is effective. Here again, the return value is zero or a negative error number. These functions are internally mapped on top of lower level callback functions exposed by the driver of the GPIO controller that provides the GPIO we use. In the next [Chapter 15, GPIO Controller Drivers - *gpio_chip*](#), dealing with GPIO controller drivers, we will see that a GPIO controller, through its `struct gpio_chip` structure, must expose a generic set of callback functions to use its GPIOs.

Some GPIO controllers offer the possibility to change the GPIO debounce-

interval (this is only useful when the GPIO line is configured as input). This feature is platform-dependent. One can use `int gpio_set_debounce()` to achieve that:

```
| static int gpio_set_debounce(unsigned gpio, unsigned debounce)
```

where `debounce` is the debounce time in ms.



All the preceding functions should be called in a context that may sleep. It is a good practice to claim and configure GPIOs from within the driver's `probe` function.

Accessing the GPIO – getting/setting the value

You should pay attention when accessing GPIO. In an atomic context, especially in an interrupt handler, one has to be sure the GPIO controller callback functions will not sleep. A well-designed controller driver should be able to inform other drivers (actually clients) whether call to its methods may sleep or not. This can be checked with `gpio_cansleep()` function.



*None of the functions used to access GPIO return an error code.
That is why you should pay attention and check return values
during GPIO allocation and configuration.*


```
static int gpio_get_value(unsigned gpio)
```

```
void gpio_set_value(unsigned int gpio, int value);
```

`gpio_get_value()` should be used when the GPIO is configured as input (using `gpio_direction_input()`), and return the actual value (state) of the GPIO. On the other hand, `gpio_set_value()` will affect the value of the GPIO, which should have been configured as an output using `gpio_direction_output()`. For both function, value can be considered as Boolean, where zero means low, and non-zero value mean high.

```
static int gpio_get_value_cansleep(unsigned gpio); void  
gpio_set_value_cansleep(unsigned gpio, int value);
```

They behave exactly like accessors without the `_cansleep()` name suffix, with the only difference being that they prevent the kernel from printing warnings when the GPIOs are accessed.

```
int gpio_to_irq(unsigned gpio);
```

```
static irqreturn_t my_interrupt_handler(int irq, void *dev_id) {
```

```
    [...]
```

```
    return IRQ_HANDLED; }
```

```
[...]
```

```
int gpio_int = of_get_gpio(np, 0); int irq_num =
```

```
gpio_to_irq(gpio_int); int error =
```

```
devm_request_threaded_irq(&client->dev, irq_num, NULL,
```

```
my_interrupt_handler, IRQF_TRIGGER_RISING |
```

```
IRQF_ONESHOT, input_dev->name, my_data_struct); if (error) {
```

```
    dev_err(&client->dev, "irq %d requested failed, %d\n", client->irq,  
error); return error; }
```

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

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/gpio.h> /* For Legacy integer based GPIO */

#include <linux/interrupt.h> /* For IRQ */


static unsigned int GPIO_LED_RED = 49;

static unsigned int GPIO_BTN1 = 115;

static unsigned int GPIO_BTN2 = 116;

static unsigned int GPIO_LED_GREEN = 120; static unsigned int irq;


static irq_handler_t btn1_pushed_irq_handler(unsigned int irq, void
*dev_id, struct pt_regs *regs) {

    int state;


    /* read BTN2 value and change the led state */

    state = gpio_get_value(GPIO_BTN2);
    gpio_set_value(GPIO_LED_RED, state);
    gpio_set_value(GPIO_LED_GREEN, state);
```

```
pr_info("GPIO_BTN1 interrupt: Interrupt! GPIO_BTN2 state is
%d)\n", state); return IRQ_HANDLED; }
```

```
static int __init helloworld_init(void) {
```

```
int retval;
```

```
/*
```

```
* One could have checked whether the GPIO is valid on the
controller or not, * using gpio_is_valid() function.
```

```
* Ex:
```

```
* if (!gpio_is_valid(GPIO_LED_RED)) {
```

```
* pr_infor("Invalid Red LED\n"); * return -ENODEV; * }
```

```
*/
```

```
gpio_request(GPIO_LED_GREEN, "green-led");
gpio_request(GPIO_LED_RED, "red-led");
gpio_request(GPIO_BTN1, "button-1"); gpio_request(GPIO_BTN2,
"button-2");
```

```
/*
```

```
* Configure Button GPIOs as input *
```

```
* After this, one can call gpio_set_debounce() * only if the
controller has the feature *
```

```

    * For example, to debounce a button with a delay of 200ms *
    gpio_set_debounce(GPIO_BTN1, 200); */

    gpio_direction_input(GPIO_BTN1);
    gpio_direction_input(GPIO_BTN2);

    /*

    * Set LED GPIOs as output, with their initial values set to 0

    */

    gpio_direction_output(GPIO_LED_RED, 0);
    gpio_direction_output(GPIO_LED_GREEN, 0);

    irq = gpio_to_irq(GPIO_BTN1); retval = request_threaded_irq(irq,
NULL, \

    btn1_pushed_irq_handler, \

    IRQF_TRIGGER_LOW | IRQF_ONESHOT, \

    "device-name", NULL);

    pr_info("Hello world!\n"); return 0;

}

static void __exit hellowolrd_exit(void) {

    free_irq(irq, NULL); gpio_free(GPIO_LED_RED);
    gpio_free(GPIO_LED_GREEN); gpio_free(GPIO_BTN1);
    gpio_free(GPIO_BTN2);

```

```
pr_info("End of the world\n"); }
```

```
module_init(hellowolrd_init);
```

```
module_exit(hellowolrd_exit);
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");  
MODULE_LICENSE("GPL");
```

```
struct gpio_desc {  
  
    struct gpio_chip *chip; unsigned long flags; const char *label; };  
  
#include <linux/gpio/consumer.h>
```

With the descriptor-based interface, prior to allocating and taking the ownership of GPIOs, those GPIOs must have been mapped somewhere. By mapped, I mean they should be assigned to your device, whereas with the legacy integer-based interface, you just have to fetch a number anywhere and request it as a GPIO. Actually, there are three kinds of mapping in the kernel:

- **Platform data mapping:** The mapping is done in the board file.
- **Device tree:** The mapping is done in DT style, the same as discussed in the preceding sections. This is the mapping we will discuss in this book.
- **Advanced Configuration and Power Interface mapping (ACPI):** The mapping is done in ACPI style. Generally used on x86-based systems.


```

/* gpio suffixes used for ACPI and device tree lookup */

static const char * const gpio_suffixes[] = { "gpios", "gpio" };

static struct gpio_desc *of_find_gpio(struct device *dev, const char
*con_id, unsigned int idx, enum gpio_lookup_flags *flags) {

    char prop_name[32]; /* 32 is max size of property name */

    enum of_gpio_flags of_flags; struct gpio_desc *desc; unsigned int i;

    for (i = 0; i < ARRAY_SIZE(gpio_suffixes); i++) {

        if (con_id)

            snprintf(prop_name, sizeof(prop_name), "%s-%s", con_id,

gpio_suffixes[i]); else

            snprintf(prop_name, sizeof(prop_name), "%s", gpio_suffixes[i]);

        desc = of_get_named_gpiod_flags(dev->of_node, prop_name, idx,

&of_flags);

        if (!IS_ERR(desc) || (PTR_ERR(desc) == -EPROBE_DEFER))
            break;

    }

    if (IS_ERR(desc)) return desc;

```

```

    if (of_flags & OF_GPIO_ACTIVE_LOW) *flags |=
GPIO_ACTIVE_LOW;

    return desc;

}

foo_device {

    compatible = "acme,foo"; [...]

    led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */

    <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */

    <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */


    power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>; reset-gpios =
<&gpio 1 GPIO_ACTIVE_LOW>; };

```

This is what a mapping should look like, with meaningful name.

Allocating and using GPIO

One can use either `gpiod_get()` or `gpiod_get_index()` to allocate a GPIO descriptor:

```
struct gpio_desc *gpiod_get_index(struct device *dev,  
                                  const char *con_id,  
                                  unsigned int idx,  
                                  enum gpiod_flags flags)  
struct gpio_desc *gpiod_get(struct device *dev,  
                             const char *con_id,  
                             enum gpiod_flags flags)
```

On error, these functions will return `-ENOENT` if no GPIO with the given function is assigned, or another error on which one can use the `IS_ERR()` macro. The first function returns the GPIO descriptor structure that corresponds to the GPIO at a given index, whereas the second function returns the GPIO at index 0 (useful for one-GPIO mapping). `dev` is the device to which the GPIO descriptor will belong. It is your device. `con_id` is the function within the GPIO consumer. It corresponds to the `<name>` prefix of the property name in the DT. `idx` is the index (starting from 0) of the GPIO for which one needs a descriptor. `flags` is an optional parameter that determines the GPIO initialization flags, to configure direction and/or output value. It is an instance of `enum gpiod_flags`, defined in `include/linux/gpio/consumer.h`:

```
enum gpiod_flags {  
    GPIOD_ASIS = 0,  
    GPIOD_IN = GPIOD_FLAGS_BIT_DIR_SET,  
    GPIOD_OUT_LOW = GPIOD_FLAGS_BIT_DIR_SET |  
                    GPIOD_FLAGS_BIT_DIR_OUT,  
    GPIOD_OUT_HIGH = GPIOD_FLAGS_BIT_DIR_SET |  
                     GPIOD_FLAGS_BIT_DIR_OUT |  
                     GPIOD_FLAGS_BIT_DIR_VAL,  
};
```

Now let us allocate GPIO descriptors for mappings defined in the preceding DT:

```
struct gpio_desc *red, *green, *blue, *power;  
  
red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);  
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);  
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);  
  
power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```

The LED GPIOs will be active-high, while the power GPIO will be active-low (that is, `gpiod_is_active_low(power)` will be true). The reverse operation of allocation

is done with the `gpiod_put()` function:

```
| gpiod_put(struct gpio_desc *desc);
```

Let us see how one can release `red` and `blue` GPIO LEDs:

```
| gpiod_put(blue);  
| gpiod_put(red);
```

Before we go further, keep in mind that apart from the `gpiod_get()/gpiod_get_index()` and `gpiod_put()` functions, which completely differ from `gpio_request()` and `gpio_free()`, one can perform API translation from integer-based interfaces to descriptor-based ones just by changing the `gpio_` prefix into `gpiod_`.

That said, to change direction, one should use the `gpiod_direction_input()` and `gpiod_direction_output()` functions:

```
| int gpiod_direction_input(struct gpio_desc *desc);  
| int gpiod_direction_output(struct gpio_desc *desc, int value);
```

`value` is the state to apply to the GPIO once the direction is set to output. If the GPIO controller has this feature, one can set the debounce timeout of a given GPIO using its descriptor:

```
| int gpiod_set_debounce(struct gpio_desc *desc, unsigned debounce);
```

In order to access a GPIO given its descriptor, the same attention must be paid as with the integer-based interface. In other words, one should take care whether one is in an atomic (cannot sleep) or non-atomic context, and then use the appropriate function:

```
| int gpiod_cansleep(const struct gpio_desc *desc);  
  
/* Value get/set from sleeping context */  
int gpiod_get_value_cansleep(const struct gpio_desc *desc);  
void gpiod_set_value_cansleep(struct gpio_desc *desc, int value);  
  
/* Value get/set from non-sleeping context */  
int gpiod_get_value(const struct gpio_desc *desc);  
void gpiod_set_value(struct gpio_desc *desc, int value);
```

For a GPIO descriptor mapped to IRQ, one can use `gpiod_to_irq()` in order to get the IRQ number that corresponds to the given GPIO descriptor, which can be used with the `request_irq()` function:

```
| int gpiod_to_irq(const struct gpio_desc *desc);
```

At any given time in the code, one can switch from the descriptor-based interface to the legacy integer-based interface and vice versa, using the `desc_to_gpio()` or `gpio_to_desc()` functions:

```
/* Convert between the old gpio_ and new gpiod_ interfaces */
struct gpio_desc *gpio_to_desc(unsigned gpio);
int desc_to_gpio(const struct gpio_desc *desc);
```

```

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/platform_device.h> /* For platform devices */

#include <linux/gpio/consumer.h> /* For GPIO Descriptor */

#include <linux/interrupt.h> /* For IRQ */

#include <linux/of.h> /* For DT*/

/*

* Let us consider the below mapping in device tree: *

* foo_device {

* compatible = "packt,gpio-descriptor-sample"; * led-gpios =
<&gpio2 15 GPIO_ACTIVE_HIGH>, // red * <&gpio2 16
GPIO_ACTIVE_HIGH>, // green *

* btn1-gpios = <&gpio2 1 GPIO_ACTIVE_LOW>; * btn2-gpios =
<&gpio2 31 GPIO_ACTIVE_LOW>; * };

*/

static struct gpio_desc *red, *green, *btn1, *btn2; static unsigned int
irq;

```

```
static irq_handler_t btn1_pushed_irq_handler(unsigned int irq, void
*dev_id, struct pt_regs *regs) {
```

```
    int state;
```

```
    /* read the button value and change the led state */
```

```
    state = gpiod_get_value(btn2); gpiod_set_value(red, state);
    gpiod_set_value(green, state);
```

```
    pr_info("btn1 interrupt: Interrupt! btn2 state is %d\n", state);
```

```
    return IRQ_HANDLED; }
```

```
static const struct of_device_id gpiod_dt_ids[] = {
```

```
    { .compatible = "packt,gpio-descriptor-sample", }, { /* sentinel */ }
```

```
};
```

```
static int my_pdrv_probe (struct platform_device *pdev) {
```

```
    int retval;
```

```
    struct device *dev = &pdev->dev;
```

```
    /*
```

* We use `gpiod_get/gpiod_get_index()` along with the flags * in order to configure the GPIO direction and an initial * value in a single function call.

*

* One could have used: * `red = gpiod_get_index(dev, "led", 0);` * `gpiod_direction_output(red, 0);` */

`red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_LOW); green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_LOW);`

/*

* Configure GPIO Buttons as input *

* After this, one can call `gpiod_set_debounce()` * only if the controller has the feature * For example, to debounce a button with a delay of 200ms * `gpiod_set_debounce(btn1, 200);` */

`btn1 = gpiod_get(dev, "led", 0, GPIOD_IN); btn2 = gpiod_get(dev, "led", 1, GPIOD_IN);`

`irq = gpiod_to_irq(btn1); retval = request_threaded_irq(irq, NULL, \`

`btn1_pushed_irq_handler, \`

`IRQF_TRIGGER_LOW | IRQF_ONESHOT, \`

`"gpio-descriptor-sample", NULL); pr_info("Hello! device probed!\n"); return 0;`

}


```
static void my_pdrv_remove(struct platform_device *pdev) {

    free_irq(irq, NULL); gpiod_put(red);

    gpiod_put(green); gpiod_put(btn1);

    gpiod_put(btn2);

    pr_info("good bye reader!\n"); }


static struct platform_driver mypdrv = {

    .probe = my_pdrv_probe, .remove = my_pdrv_remove, .driver = {

        .name = "gpio_descriptor_sample", .of_match_table =
of_match_ptr(gpiod_dt_ids), .owner = THIS_MODULE, },

};

module_platform_driver(mypdrv);

MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
MODULE_LICENSE("GPL");
```

```
gpio1: gpio1 {  
    gpio-controller; #gpio-cells = <2>; };
```

```
gpio2: gpio2 {  
    gpio-controller; #gpio-cells = <1>; };
```

```
[...]
```

```
cs-gpios = <&gpio1 17 0>, <&gpio2 2>; <0>, /* holes are permitted,  
means no GPIO 2 */
```

```
<&gpio1 17 0>;
```

```
reset-gpios = <&gpio1 30 0>; cd-gpios = <&gpio2 10>;
```

In the preceding sample, CS GPIOs contain both controller-1 and controller-2 GPIOs. If one does not need to specify a GPIO at a given index in the list, one can use <0>. The reset GPIO has two cells (two arguments after the controller phandle), whereas CD GPIO has only one cell. You can see how meaningful the names are that I gave to my GPIO specifier.

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

```
int of_get_named_gpio(struct device_node *np, const char  
*propname, int index) int of_get_named_gpio_count(struct  
device_node *np, const char* propname)
```

```
int n_gpios = of_get_named_gpio_count(dev.of_node, "cs-gpios"); /*  
return 4 */
```

```
int second_gpio = of_get_named_gpio(dev.of_node, "cs-gpio", 1); int  
rst_gpio = of_get_named_gpio("reset-gpio", 0);  
gpio_request(second_gpio, "my-gpio");
```

```
int of_gpio_count(struct device_node *np) int of_get_gpio(struct  
device_node *np, int index)
```

```
my_node@addr {
```

```
    compatible = "[...]";
```

```
    gpios = <&gpio1 2 0>, /* INT */
```

```
    <&gpio1 5 0>; /* RST */
```

```
    [...]
```

```
};
```

```
struct device_node *np = dev->of_node;
```

```
if (!np)
```

```
    return ERR_PTR(-ENOENT);
```

```

int n_gpios = of_gpio_count(); /* Will return 2 */

int gpio_int = of_get_gpio(np, 0);

if (!gpio_is_valid(gpio_int)) {

    dev_err(dev, "failed to get interrupt gpio\n"); return ERR_PTR(-
EINVAL); }

gpio_rst = of_get_gpio(np, 1);

if (!gpio_is_valid(pdata->gpio_rst)) {

    dev_err(dev, "failed to get reset gpio\n"); return ERR_PTR(-
EINVAL); }

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/platform_device.h> /* For platform devices */

#include <linux/interrupt.h> /* For IRQ */

#include <linux/gpio.h> /* For Legacy integer based GPIO */

#include <linux/of_gpio.h> /* For of_gpio* functions */

#include <linux/of.h> /* For DT*/

```

```
/*
```

```
* Let us consider the following node
```

```
*
```

```
* foo_device {
```

```
* compatible = "packt,gpio-legacy-sample"; * led-gpios = <&gpio2  
15 GPIO_ACTIVE_HIGH>, // red * <&gpio2 16  
GPIO_ACTIVE_HIGH>, // green *
```

```
* btn1-gpios = <&gpio2 1 GPIO_ACTIVE_LOW>; * btn2-gpios =  
<&gpio2 1 GPIO_ACTIVE_LOW>; * };
```

```
*/
```

```
static unsigned int gpio_red, gpio_green, gpio_btn1, gpio_btn2; static  
unsigned int irq;
```

```
static irq_handler_t btn1_pushed_irq_handler(unsigned int irq, void  
*dev_id, struct pt_regs *regs) {
```

```
    /* The content of this function remains unchanged */
```

```
    [...]
```

```
}
```

```

static const struct of_device_id gpio_dt_ids[] = {

    { .compatible = "packt,gpio-legacy-sample", }, { /* sentinel */ }

};

static int my_pdrv_probe (struct platform_device *pdev) {

    int retval;

    struct device_node *np = &pdev->dev.of_node;

    if (!np)

        return ERR_PTR(-ENOENT);

    gpio_red = of_get_named_gpio(np, "led", 0); gpio_green =
of_get_named_gpio(np, "led", 1); gpio_btn1 =
of_get_named_gpio(np, "btn1", 0); gpio_btn2 =
of_get_named_gpio(np, "btn2", 0);

    gpio_request(gpio_green, "green-led"); gpio_request(gpio_red,
"red-led"); gpio_request(gpio_btn1, "button-1");
gpio_request(gpio_btn2, "button-2");

    /* Code to configure GPIO and request IRQ remains unchanged */

    [...]

    return 0;

}

```

```
static void my_pdrv_remove(struct platform_device *pdev) {  
  
    /* The content of this function remains unchanged */  
  
    [...]  
  
}
```

```
static struct platform_driver mypdrv = {  
  
    .probe = my_pdrv_probe, .remove = my_pdrv_remove, .driver = {  
  
        .name = "gpio_legacy_sample", .of_match_table =  
of_match_ptr(gpio_dt_ids), .owner = THIS_MODULE, },  
  
};
```

```
module_platform_driver(mypdrv);
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");  
MODULE_LICENSE("GPL");
```

```
gpio4: gpio4 {
```

```
    gpio-controller; #gpio-cells = <2>; interrupt-controller; #interrupt-  
cells = <2>; };
```

```
my_label: node@0 {
```

```
    reg = <0>; spi-max-frequency = <1000000>; interrupt-parent =  
<&gpio4>; interrupts = <29 IRQ_TYPE_LEVEL_LOW>; };
```

```
int platform_get_irq(struct platform_device *dev, unsigned int num);
```

Feel free to have a look at [Chapter 6](#), *The Concept of Device Tree*.

GPIO and sysfs

The sysfs GPIO interface lets people manage and control GPIOs through sets or files. It is located under `/sys/class/gpio`. The device model is heavily used here, and there are three kinds of entries available:

- `/sys/class/gpio/`: This is where everything begins. This directory contains two special files, `export` and `unexport`:
 - `export`: This allow us to ask the kernel to export control of a given GPIO to user space by writing its number to this file. Example: `echo 21 > export` will create a `GPIO21` node for GPIO #21, if that's not requested by kernel code.
 - `unexport`: This reverses the effect of exporting to user space. Example: `echo 21 > unexport` will remove any `GPIO21` node exported using the `export` file.
- `/sys/class/gpio/gpioN/`: This directory corresponds to the GPIO number N (where N is global to the system, not relative to the chip), exported either using the `export` file, or from within the kernel. For example:
`/sys/class/gpio/gpio42/` (for GPIO #42) with the following read/write attributes:
 - The `direction` file is used to get/set GPIO direction. Allowed values are either `in` or `out` strings. This value may normally be written. Writing as `out` defaults to initializing the value as low. To ensure glitch-free operation, low and high values may be written to configure the GPIO as an output with that initial value. This attribute will not exist if the kernel code has exported this GPIO, disabling direction (see the `gpiod_export()` OR `gpio_export()` function).
 - The `value` attribute lets us get/set the state of the GPIO line, depending on the direction, input or output. If the GPIO is configured as an output, any non-zero value written will be treated as HIGH state. If

configured as an output, writing `0` will set the output low, whereas `1` will set the output high. If the pin can be configured as an interrupt-generating lines and if it has been configured to generate, one can call the `poll(2)` system call on that file and `poll(2)` will return whenever the interrupt was triggered. Using `poll(2)` will require setting the events `POLLPRI` and `POLLERR`. If one uses `select(2)` instead, one should set the file descriptor in `exceptfds`. After `poll(2)` returns, either `lseek(2)` to the beginning of the sysfs file and read the new value or close the file and re-open it to read the value. It is the same principle as we discussed regarding the pollable sysfs attribute.

- `edge` determines the signal edge that will let the `poll()` or `select()` function return. Allowed values are `none`, `rising`, `falling`, or `both`. This file is readable/writable, and exists only if the pin can be configured as an interrupt-generating input pin.
- `active_low` reads as either `0` (false) or `1` (true). Writing any non-zero value will invert the *value* attribute for both reading and writing. Existing and subsequent `poll(2)` support configurations through the edge attribute for rising and falling edges will follow this setting. The relevant function from kernel to set this value is `gpio_sysfs_set_active_low()`.

```

int gpio_export(unsigned gpio, bool direction_may_change);

int gpiod_export(struct gpio_desc *desc, bool direction_may_change);

void gpio_unexport(unsigned gpio); /* Integer-based interface */

void gpiod_unexport(struct gpio_desc *desc) /* Descriptor-based */

int gpio_export_link(struct device *dev, const char *name, unsigned
gpio)

int gpiod_export_link(struct device *dev, const char *name, struct
gpio_desc *desc)

static struct gpio_desc *red, *green, *btn1, *btn2;

static int my_pdrv_probe (struct platform_device *pdev) {

    [...]

    red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_LOW); green =
gpiod_get_index(dev, "led", 1, GPIOD_OUT_LOW);

    gpiod_export(&pdev->dev, "Green_LED", green);
gpiod_export(&pdev->dev, "Red_LED", red);

    [...]

    return 0;

}

static int my_pdrv_probe (struct platform_device *pdev) {

    [...]

```

```
    gpio_red = of_get_named_gpio(np, "led", 0); gpio_green =  
of_get_named_gpio(np, "led", 1); [...]
```

```
    int gpio_export_link(&pdev->dev, "Green_LED", gpio_green) int  
gpio_export_link(&pdev->dev, "Red_LED", gpio_red) return 0;  
}
```

Summary

Dealing with a GPIO from within the kernel is an easy task, as shown in this chapter. Both legacy and new interfaces are discussed, giving the possibility to choose the one that fits your needs, in order to write enhanced GPIO drivers. You'll be able to handle IRQs mapped to GPIOs. The next chapter will deal with the chip that provides and exposes GPIO lines, known as the GPIO controller.

GPIO Controller Drivers – gpio_chip

In the previous chapter, we dealt with GPIO lines. Those lines are exposed to the system by means of a special device called the GPIO controller. This chapter will explain step by step how to write drivers for such devices, thus covering the following topics:

- GPIO controller driver architecture and data structures
- Sysfs interface for GPIO controllers
- GPIO controllers representation in DT

```

struct gpio_chip {

    const char *label;

    struct device *dev;

    struct module *owner;

    int (*request)(struct gpio_chip *chip, unsigned offset); void (*free)
(struct gpio_chip *chip, unsigned offset); int (*get_direction)(struct
gpio_chip *chip, unsigned offset); int (*direction_input)(struct
gpio_chip *chip, unsigned offset); int (*direction_output)(struct
gpio_chip *chip, unsigned offset, <br/> int value); int (*get)(struct
gpio_chip *chip, unsigned offset); void (*set)(struct gpio_chip *chip,
unsigned offset, int value); void (*set_multiple)(struct gpio_chip
*chip, unsigned long *mask, unsigned long *bits);

    int (*set_debounce)(struct gpio_chip *chip, unsigned offset,
unsigned debounce);

    int (*to_irq)(struct gpio_chip *chip, unsigned offset);

    int base;

    u16 ngpio;

    const char *const *names; bool can_sleep;

    bool irq_not_threaded; bool exported;

```

```
#ifndef CONFIG_GPIOLIB_IRQCHIP
```

```
/*
```

```
 * With CONFIG_GPIOLIB_IRQCHIP we get an irqchip * inside  
the gpiolib to handle IRQs for most practical cases.
```

```
*/
```

```
struct irq_chip *irqchip; struct irq_domain *irqdomain; unsigned int  
irq_base; irq_flow_handler_t irq_handler; unsigned int  
irq_default_type; #endif
```

```
#if defined(CONFIG_OF_GPIO)
```

```
/*
```

```
 * If CONFIG_OF is enabled, then all GPIO controllers described in  
the * device tree automatically may have an OF translation */
```

```
struct device_node *of_node; int of_gpio_n_cells;
```

```
int (*of_xlate)(struct gpio_chip *gc, const struct of_phandle_args  
*gpiospec, u32 *flags); }
```

```
static void gpio_chip_set_multiple(struct gpio_chip *chip, unsigned  
long *mask, unsigned long *bits) {
```

```
if (chip->set_multiple) {
```

```
chip->set_multiple(chip, mask, bits); } else {
```



```

unsigned int i;

/* set outputs if the corresponding mask bit is set */

for_each_set_bit(i, mask, chip->ngpio) chip->set(chip, i, test_bit(i,
bits)); }

}

#include <linux/gpio.h>

#define GPIO_NUM 16

struct mcp23016 {

    struct i2c_client *client; struct gpio_chip chip; };

static int mcp23016_probe(struct i2c_client *client, const struct
i2c_device_id *id) {

    struct mcp23016 *mcp;

    if (!i2c_check_functionality(client->adapter,
I2C_FUNC_SMBUS_BYTE_DATA)) return -EIO;

    mcp = devm_kzalloc(&client->dev, sizeof(*mcp), GFP_KERNEL);
if (!mcp)

```

```
return -ENOMEM;
```

```
mcp->chip.label = client->name; mcp->chip.base = -1; mcp->chip.dev = &client->dev; mcp->chip.owner = THIS_MODULE; mcp->chip.ngpio = GPIO_NUM; /* 16 */
```

```
mcp->chip.can_sleep = 1; /* may not be accessed from atomic context */
```

```
mcp->chip.get = mcp23016_get_value; mcp->chip.set = mcp23016_set_value; mcp->chip.direction_output = mcp23016_direction_output; mcp->chip.direction_input = mcp23016_direction_input; mcp->client = client; i2c_set_clientdata(client, mcp);
```

```
return gpiochip_add(&mcp->chip); }
```

```
struct gpio_desc *gpiochip_request_own_desc(struct gpio_desc *desc, const char *label) void gpiochip_free_own_desc(struct gpio_desc *desc)
```

Descriptors requested with `gpiochip_request_own_desc()` must be released with `gpiochip_free_own_desc()`.

Pin controller guideline

Depending on the controller you write the driver for, you may need to implement some pin control operation to handle pin multiplexing, configuration, and so on:

- For a pin controller that can only do simple GPIO, a simple `struct gpio_chip` will be sufficient to handle it. There is no need to set up a `struct pinctrl_desc` structure, just write the GPIO controller driver as it.
- If the controller can generate interrupts on top of the GPIO functionality, a `struct irq_chip` must be set up and registered to the IRQ subsystem.
- For a controller having pin multiplexing, advanced pin driver strength, complex biasing, you should set up the following three interfaces :
 - `struct gpio_chip`, discussed earlier in this chapter
 - `struct irq_chip`, discussed in the next chapter ([Chapter 16](#), *Advanced IRQ Management*)
 - `struct pinctrl_desc`, not discussed in the book, but well explained in the kernel documentation in *Documentation/pinctrl.txt*

Sysfs interface for GPIO controller

On successful `gpiochip_add()`, a directory entry with a path like `/sys/class/gpio/gpiochipX/` will be created, where `x` is the GPIO controller base (controller providing GPIOs starting at `#x`), having the following attributes:

- `base`, whose value is same as `x`, and which corresponds to `gpio_chip.base` (if assigned statically), and being the first GPIO managed by this chip.
- `label`, which is provided for diagnostics (not always unique).
- `ngpio`, which tells how many GPIOs this controller provides (`N` to `N + ngpio - 1`). This is the same as defined in `gpio_chip.ngpios`.

All of the preceding attributes are read-only.

```
expander_1: mcp23016@27 {
```

```
    compatible = "microchip,mcp23016"; interrupt-controller; gpio-  
controller;
```

```
    #gpio-cells = <2>; interrupt-parent = <&gpio6>; interrupts = <31  
IRQ_TYPE_LEVEL_LOW>; reg = <0x27>; #interrupt-cells=<2>; };
```

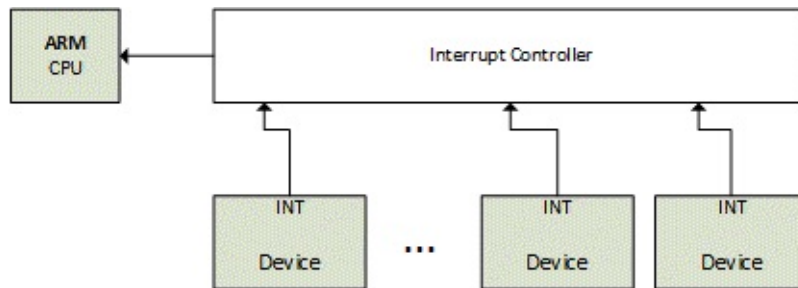
The preceding sample is the node of our GPIO-controller device, and the complete device driver is provided with the sources of the book.

Summary

This chapter is much more than a basis to write the driver for a GPIO controller that you may encounter. It explains the main structure to describe such devices. The next chapter deals with advanced IRQ management, in which we will see how to manage an interrupt controller and thus add such functionality in the driver of the MCP23016 expander from microchip.

Advanced IRQ Management

Linux is a system on which devices notify the kernel about particular events by means of IRQs. The CPU exposes IRQ lines, shared or not, and used by connected devices, so that when a device needs the CPU it sends a request to the CPU. When the CPU gets this request it stops its actual job and saves its context, in order to serve the request issued by the device. After serving the device, its state is restored back to exactly where it stopped when the interruption occurred. There are so many IRQ lines, that another device is responsible for them to the CPU. That device is the interrupt controller:



Interrupt controller and IRQ lines

Not only can devices raise interrupts, some processor operations can do that too. There are two different kinds of interrupts:

1. Synchronous interrupts called **exceptions**, produced by the CPU while processing instructions. These are **non-maskable interrupts (NMI)**, and result from a critical malfunction such as hardware failure. They are always processed by the CPU.
2. Asynchronous interrupts called **interrupts**, are issued by other hardware devices. These are normal and maskable interrupts. It is what we will discuss in the next sections of this chapter. Therefore, let us go a bit deeper into exceptions:

Exceptions are consequences of programming errors handled by the kernel, which sends a signal to the program and tries to recover from the error. These are classified in two categories, enumerated below:

- **Processor-detected exceptions:** Those the CPU generates in response to an anomalous condition, and it is divided into three groups:
 - Faults, which can generally be corrected (bogus instruction).
 - Traps, which occur in user process (invalid memory access, division by zero), are also a mechanism to switch to kernel mode in response to a system call. If the kernel code does cause a trap, it immediately panics.
 - Aborts, the serious errors.
- **Programmed exception:** These are requested by the programmer, handled like a trap.

The following array lists unmaskable interrupts (for more details refer to <http://wiki.osdev.org/Exceptions>):

Interrupt number	Description
0	Divide by zero error
1	Debug exception
2	NMI interrupt
3	Breakpoint
4	INTO detected overflow
5	BOUND range exceeded
6	Invalid opcode

7	Coprocessor (device) not available
8	Double fault
9	Coprocessor segment overrun
10	Invalid task state segment
11	Segment not present
12	Stack fault
13	General protection fault
14	Page fault
15	Reserved
16	Coprocessor error
17 - 31	Reserved
32 - 255	Maskable interrupts

NMIs are enough to cover the whole exception list. Back to maskable interrupts,

their number depends on the number of devices connected, and how they actually share those IRQ lines. Sometimes, they are not enough and some of them need multiplexing. The commonly used method is by means of a GPIO controller, which also acts as an interrupt controller. In this chapter, we will deal with the API that the kernel offers to manage IRQ and the ways by which multiplexing can be done, and get deeper in interrupt controller driver writing.

That said, in this chapter the following topics will be covered:

- Interrupt controllers and interrupt multiplexing
- Advanced peripheral IRQs management
- Interrupt requests and propagations (chained or nested)
- GPIOLIB irqchip API
- Handling interrupt controllers from DT

```

struct irq_chip {

    struct device *parent_device; const char *name;

    void (*irq_enable)(struct irq_data *data); void (*irq_disable)(struct
irq_data *data);

    void (*irq_ack)(struct irq_data *data); void (*irq_mask)(struct
irq_data *data); void (*irq_unmask)(struct irq_data *data); void
(*irq_eoi)(struct irq_data *data);

    int (*irq_set_affinity)(struct irq_data *data, const struct cpumask
*dest, bool force); int (*irq_retrigger)(struct irq_data *data); int
(*irq_set_type)(struct irq_data *data, unsigned int flow_type); int
(*irq_set_wake)(struct irq_data *data, unsigned int on);

    void (*irq_bus_lock)(struct irq_data *data); void
(*irq_bus_sync_unlock)(struct irq_data *data);

    int (*irq_get_irqchip_state)(struct irq_data *data, enum
irqchip_irq_state which, bool *state); int(*irq_set_irqchip_state)
(struct irq_data *data, enum irqchip_irq_state which, bool state);

    unsigned long flags; };

struct irq_domain {

    const char *name;

    const struct irq_domain_ops *ops; void *host_data;

    unsigned int flags;

    /* Optional data */

```

```

    struct fwnode_handle *fwnode; [...]

};

unsigned int irq_create_mapping(struct irq_domain *domain,
irq_hw_number_t hwirq)

for (j = 0; j < gpiochip->chip.ngpio; j++) {

    irq = irq_create_mapping(

        gpiochip ->irq_domain, j); }

struct irq_domain_ops {

    int (*map)(struct irq_domain *d, unsigned int virq,
irq_hw_number_t hw); void (*unmap)(struct irq_domain *d, unsigned
int virq); int (*xlate)(struct irq_domain *d, struct device_node *node,
const u32 *intspec, unsigned int intsize, unsigned long *out_hwirq,
unsigned int *out_type); };

static int pcf857x_irq_domain_map(struct irq_domain *domain,
unsigned int irq, irq_hw_number_t hw) {

    struct pcf857x *gpio = domain->host_data;

    irq_set_chip_and_handler(irq,
&dummy_irq_chip,handle_level_irq); #ifdef CONFIG_ARM

    set_irq_flags(irq, IRQF_VALID); #else

    irq_set_noprobe(irq); #endif

    gpio->irq_mapped |= (1 << hw);

```

```

return 0;

}

static struct irq_domain_ops mcp23016_irq_domain_ops = {

    .map = mcp23016_irq_domain_map, .xlate =
irq_domain_xlate_twocell, };

struct irq_desc {

    struct irq_common_data irq_common_data; struct irq_data irq_data;
unsigned int __percpu *kstat_irqs; irq_flow_handler_t handle_irq;
struct irqaction *action; unsigned int irq_unhandled; raw_spinlock_t
lock; struct cpumask *percpu_enabled; atomic_t threads_active;
wait_queue_head_t wait_for_threads; #ifdef CONFIG_PM_SLEEP

    unsigned int nr_actions; unsigned int no_suspend_depth; unsigned
int force_resume_depth; #endif

#ifdef CONFIG_PROC_FS

    struct proc_dir_entry *dir; #endif

    int parent_irq;

    struct module *owner; const char *name;

};

struct irqaction {

    irq_handler_t handler; void *dev_id;

    void __percpu *percpu_dev_id; struct irqaction *next; irq_handler_t

```

```

thread_fn; struct task_struct *thread; unsigned int irq;

    unsigned int flags; unsigned long thread_flags; unsigned long
thread_mask; const char *name;

    struct proc_dir_entry *dir; };

struct irq_data {

    [...]

    unsigned int irq;

    unsigned long hwirq; struct irq_common_data *common; struct
irq_chip *chip; struct irq_domain *domain; void *chip_data;

};

```

- `irq` is the interrupt number (Linux IRQ)
- `hwirq` is the hardware interrupt number, local to the `irq_data.domain` interrupt domain
- `common` points to data shared by all irqchips
- `chip` represents the low level interrupt controller hardware access
- `domain` represents the interrupt translation domain, responsible for mapping between the `hwirq` number and the Linux `irq` number
- `chip_data` is a platform-specific per-chip private data for the chip methods, to allow shared chip implementations

```
int request_any_context_irq ( unsigned int irq, irq_handler_t handler,  
unsigned long flags, const char * name, void * dev_id);
```

```
static irqreturn_t packt_btn_interrupt(int irq, void *dev_id) {  
  
    struct btn_data *priv = dev_id;  
  
    input_report_key(priv->i_dev, BTN_0, gpiod_get_value(priv-  
>btn_gpiod) & 1); input_sync(priv->i_dev); return IRQ_HANDLED;  
}
```

```
static int btn_probe(struct platform_device *pdev) {
```

```
    struct gpio_desc *gpiod; int ret, irq;
```

```
    [...]
```

```
    gpiod = gpiod_get(&pdev->dev, "button", GPIOD_IN); if  
(IS_ERR(gpiod)) return -ENODEV;
```

```
    priv->irq = gpiod_to_irq(priv->btn_gpiod); priv->btn_gpiod =  
gpiod;
```

```
    [...]
```

```
    ret = request_any_context_irq(priv->irq, packt_btn_interrupt,
```

```

(IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING), "packt-
input-button", priv); if (ret < 0) {

    dev_err(&pdev->dev, "Unable to acquire interrupt for GPIO
line\n"); goto err_btn;

}

return ret;

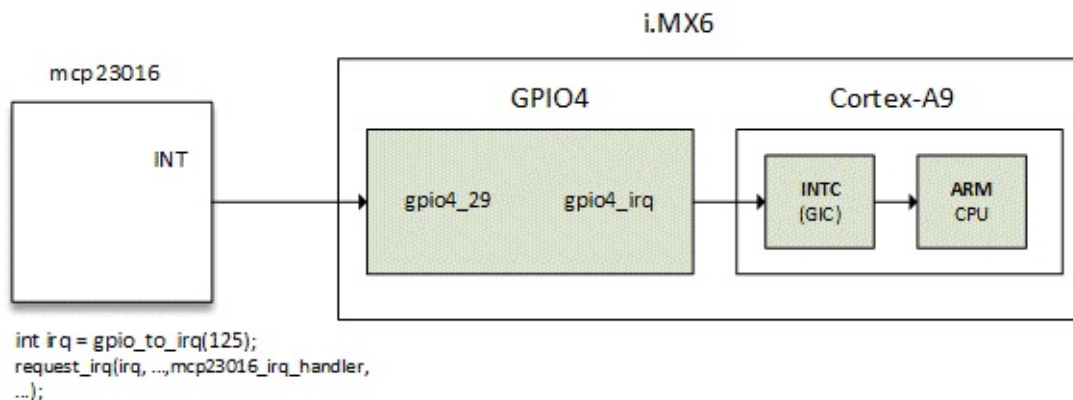
}

```

The preceding code is an excerpt of the driver sample of an input device driver. Actually, it is the one used in the next chapter. The advantage using `request_any_context_irq()` is that, one does not need to care about what can be done in the IRQ handler, as the context in which the handler will run depends on the interrupt controller that provides the IRQ line. In our example, if the GPIO below to a controller seating on an I2C or SPI bus, the handler will be threaded. Otherwise, the handler will run in hardirq.

Interrupt request and propagation

Let us consider the following figure, which represents a chained IRQ flow



Interrupt requests are always performed on Linux IRQ (not hwirq). The general function to request IRQ on Linux is `request_threaded_irq()` or `request_irq()`, which internally calls the former:

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                        irq_handler_t thread_fn, unsigned long irqflags,
                        const char *devname, void *dev_id)
```

When called, the function extracts the `struct irq_desc` associated with the IRQ using the `irq_to_desc()` macro. It then allocates a new `struct irqaction` structure and sets it up, filling parameters such as handler, flags, and so on.

```
action->handler = handler;
action->thread_fn = thread_fn;
action->flags = irqflags;
action->name = devname;
action->dev_id = dev_id;
```

That same function finally inserts/registers the descriptor in the proper IRQ list by invoking `__setup_irq()` (by means of `setup_irq()`) function, defined in `kernel/irq/manage.c`.

Now, when an IRQ is raised, the kernel executes a few assembler codes in order to save the current state, and jumps to the arch specific handler, `handle_arch_irq`, which is set with the `handle_irq` field of `struct machine_desc` of our platform in the

setup_arch() function, in arch/arm/kernel/setup.c:

```
| handle_arch_irq = mdesc->handle_irq
```

For SoCs that use ARM GIC, handle_irq callback is set with gic_handle_irq, in either drivers/irqchip/irq-gic.c, or drivers/irqchip/irq-gic-v3.c:

```
| set_handle_irq(gic_handle_irq);
```

gic_handle_irq() calls handle_domain_irq(), which executes generic_handle_irq(), its turn calling generic_handle_irq_desc() that ends by calling desc->handle_irq(). Have a look at include/linux/irqdesc.h for the last call and arch/arm/kernel/irq.c for other function calls. handle_irq is the actual call for the flow handler, which we registered as mcp23016_irq_handler.

gic_handle_irq() is a GIC interrupt handler. generic_handle_irq() will execute the handler of the SoC's GPIO4 IRQ, which will look for GPIOs pin responsible for the interrupt, and call generic_handle_irq_desc(), and so on. Now that you are familiar with interrupt propagation, let us switch to a practical example by writing our own interrupt controller.

Chaining IRQ

This section describes how interrupt handlers of a parent, call the child's interrupt handlers, in turn calling their child's interrupt handlers, and so on. The kernel offers two approaches on how to call interrupt handlers for child devices in the IRQ handler of the parent (interrupt controller) device. These are the chained and nested methods:

Chained interrupts

This approach is used for SoC's internal GPIO controller, which are memory mapped and whose access does not sleep. Chained means that those interrupts are just chains of function calls (for example, SoC's GPIO module interrupt handler is being called from GIC interrupt handler, just as a function call).

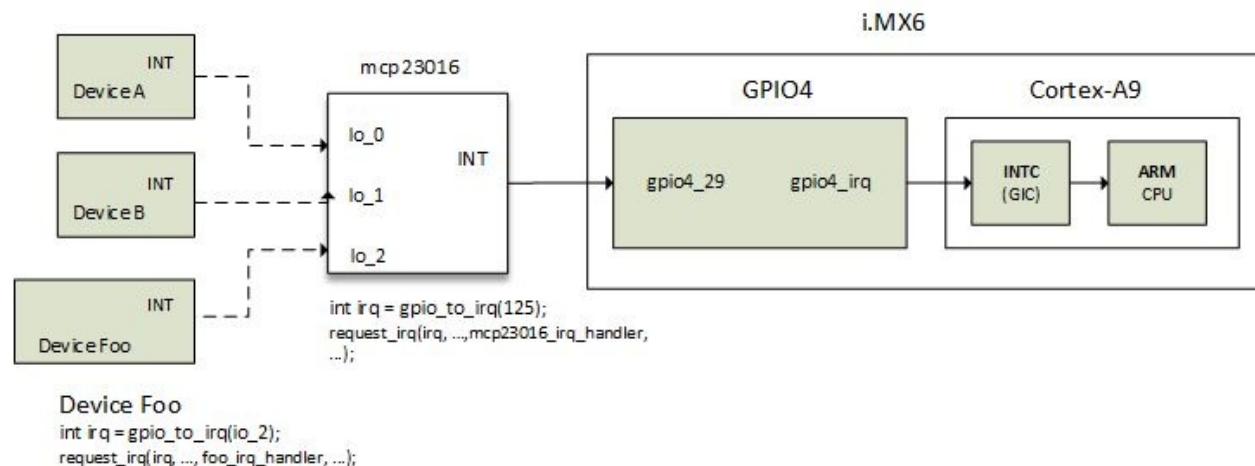
`generic_handle_irq()` is used for interrupts chaining child IRQ handlers and are called inside of the parent hwirq handler. Even from within the child interrupt handlers, we still are in atomic context (HW interrupt). One cannot call functions that may sleep.

Nested interrupts

This method is used by controllers that sit on slow buses, like I2C (for example, GPIO expander), and whose access may sleep (I2C functions may sleep). Nested means those interrupts handlers that do not run in the HW context (they are not really hwirq, they are not in atomic context), but they are threaded instead, and can be preempted (or interrupted by another interrupt). `handle_nested_irq()` is used for creating nested interrupt child IRQs. Handlers are being called inside of the new thread created by the `handle_nested_irq()` function; we need them to be run in process context, so that we can call sleeping bus functions (like I2C functions that may sleep).

Case study – GPIO and IRQ chip

Let us consider the following figure that ties an interrupt controller device to another one, which we will use to describe interrupt multiplexing:



mcp23016 IRQ flow

Consider that you have configured `io_1` and `io_2` as interrupts. Even if interrupt happens on `io_1` or `io_2`, the same interrupt line will be triggered to the interrupt controller. Now the GPIO driver has to figure out reading the interrupt status register of the GPIO to find which interrupt (`io_1` or `io_2`) has really fired. Therefore, in this case a single interrupt line is a multiplex for 16 GPIO interrupts.

Now let us mangle the original driver of the mcp23016 written in [Chapter 15](#), *GPIO Controller Drivers – gpio_chip* in order to support IRQ domain API first, which will let it act as an interrupt controller as well. The second part will introduce the new and recommended gpiolib irqchip API. This will be used as a step-by-step guide to write the interrupt controller driver, at least for the GPIO controller:

```

devm_request_threaded_irq(client->irq, NULL, mcp23016_irq,
irqflags, dev_name(chip->parent), mcp);

static irqreturn_t mcp23016_irq(int irq, void *data) {

    struct mcp23016 *mcp = data; unsigned int child_irq, i; /* Do some
stuff */

    [...]

    for (i = 0; i < mcp->chip.ngpio; i++) {

        if (gpio_value_changed_and_raised_irq(i)) {

            child_irq =

                irq_find_mapping(mcp->chip.irqdomain, i);
            handle_nested_irq(child_irq); }

        }

    return IRQ_HANDLED; }

```

handle_nested_irq() already described in the preceding section will create a dedicated thread for each handler registered.

New gpiolib irqchip API

Almost every GPIO controller driver was using IRQ domain for the same purpose. Instead of each of them rolling their own irqdomain handling and so on, kernel developers decided to move that code to gpiolib framework, by means of GPIOLIB_IRQCHIP Kconfig symbol, in order to harmonize the development and avoid redundant code.

That portion of code helps in handling management of GPIO irqchips and the associated `irq_domain` and resource allocation callbacks, as well as their setup, using the reduced set of helper functions. These are `gpiochip_irqchip_add()` and `gpiochip_set_chained_irqchip()`.

`gpiochip_irqchip_add()`: This adds an irqchip to a gpiochip. What this function does:

- Sets `gpiochip.to_irq` field to `gpiochip_to_irq`, which is an IRQ callback that just returns `irq_find_mapping(chip->irqdomain, offset)`;
- Allocates an `irq_domain` to the gpiochip using `irq_domain_add_simple()` function, passing a kernel IRQ core `irq_domain_ops` called `gpiochip_domain_ops` and defined in `drivers/gpio/gpiolib.c`
- Create mapping from 0 to `gpiochip.ngpio` using `irq_create_mapping()` function

Its prototype is as follows:

```
int gpiochip_irqchip_add(struct gpio_chip *gpiochip,
                        struct irq_chip *irqchip,
                        unsigned int first_irq,
                        irq_flow_handler_t handler,
                        unsigned int type)
```

Where `gpiochip` is our GPIO chip, the one to add the irqchip to, `irqchip` is the irqchip to add to the gpiochip. `first_irq` if not dynamically assigned, is the base (first) IRQ to allocate gpiochip IRQs from. `handler` is the IRQ handler to use (often a predefined IRQ core function), and `type` is the default type for IRQs on this irqchip, pass `IRQ_TYPE_NONE` to have the core avoid setting up any default type in the hardware.



This function will handle two celled simple IRQs (because it sets



irq_domain_ops.xlate to irq_domain_xlate_twocell) and assumes all the pins on the gpiochip can generate a unique IRQ.

```
static const struct irq_domain_ops gpiochip_domain_ops = {
    .map = gpiochip_irq_map,
    .unmap = gpiochip_irq_unmap,
    /* Virtually all GPIO irqchips are twocell:ed */
    .xlate = irq_domain_xlate_twocell,
};
```

`gpiochip_set_chained_irqchip()`: This function sets a chained irqchip to a `gpio_chip` from a parent IRQ and passes a pointer to the `struct gpio_chip` as handler data:

```
void gpiochip_set_chained_irqchip(struct gpio_chip *gpiochip,
                                struct irq_chip *irqchip, int parent_irq,
                                irq_flow_handler_t parent_handler)
```

`parent_irq` is the IRQ number to which this chip is connected. In case of our `mcp23016` as shown in the figure in the section *Case study-GPIO and IRQ chip*, it corresponds to the IRQ of `gpio4_29` line. In other words, it is the parent IRQ number for this chained irqchip. `parent_handler` is the parent interrupt handler for the accumulated IRQ coming out of the gpiochip. If the interrupt is nested rather than cascaded (chained), pass `NULL` in this handler argument.

With this new API, the only code to add to our `probe` function is:

```
/* Do we have an interrupt line? Enable the irqchip */
if (client->irq) {
    status = gpiochip_irqchip_add(&gpio->chip, &dummy_irq_chip,
                                0, handle_level_irq, IRQ_TYPE_NONE);

    if (status) {
        dev_err(&client->dev, "cannot add irqchip\n");
        goto fail_irq;
    }

    status = devm_request_threaded_irq(&client->dev, client->irq,
                                     NULL, mcp23016_irq, IRQF_ONESHOT |
                                     IRQF_TRIGGER_FALLING | IRQF_SHARED,
                                     dev_name(&client->dev), gpio);

    if (status)
        goto fail_irq;

    gpiochip_set_chained_irqchip(&gpio->chip,
                                &dummy_irq_chip, client->irq, NULL);
}
```

IRQ core does everything for us. No need to define even the `gpiochip.to_irq` function, since the API already sets it. Our example uses the IRQ core `dummy_irq_chip`, but one could have defined its own as well. Since the v4.10 version of the kernel, two other functions have been added: these are

`gpiochip_irqchip_add_nested()` and `gpiochip_set_nested_irqchip()`. Have a look at *Documentation/gpio/driver.txt* for more details. A driver that uses this API in the same kernel version is `drivers/gpio/gpio-mcp23s08.c`.

```
expander: mcp23016@20 {
```

```
    compatible = "microchip,mcp23016"; reg = <0x20>; interrupt-  
controller; #interrupt-cells = <2>; gpio-controller;
```

```
    #gpio-cells = <2>; interrupt-parent = <&gpio4>; interrupts = <29  
IRQ_TYPE_EDGE_FALLING>; };
```

```
foo_device: foo_device@1c {
```

```
    reg = <0x1c>; interrupt-parent = <&expander>; interrupts = <2  
IRQ_TYPE_EDGE_RISING>; };
```

```
bar_device {
```

```
    reset-gpios = <&expander 8 GPIO_ACTIVE_HIGH>; power-gpios  
= <&expander 12 GPIO_ACTIVE_HIGH>; /* Other properties do  
here */
```

```
};
```

Summary

Now IRQ multiplexing has no more secrets for you. We discussed the most important elements of IRQ management under Linux systems, the IRQ domain API. You have the basics to develop interrupt controller drivers, as well as managing their binding from within the DT. IRQ propagation has been discussed in order to understand what happens from the request to the handling. This chapter will help you to understand the interrupt driven part of the next chapter, which deals with input device drivers.

Input Devices Drivers

Input devices are devices by which one can interact with the system. Such devices are buttons, keyboards, touchscreens, mouse, and so on. They work by sending events, caught and broadcasted over the system by the input core. This chapter will explain each structure used by input core to handle input devices. That being said, we will see how one can manage events from the user space.

In this chapter, we will cover following topics:

- Input core data structures
- Allocating and registering input devices, and well as polled devices family
- Generating and reporting events to the input core
- Input device from user space
- Writing a driver example

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

```
struct input_dev {
```

```
    const char *name; const char *phys;
```

```
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; unsigned long  
    keybit[BITS_TO_LONGS(KEY_CNT)]; unsigned long  
    relbit[BITS_TO_LONGS(REL_CNT)]; unsigned long  
    absbit[BITS_TO_LONGS(ABS_CNT)]; unsigned long  
    mscbit[BITS_TO_LONGS(MSC_CNT)];
```

```
    unsigned int repeat_key;
```

```
    int rep[REP_CNT]; struct input_absinfo *absinfo; unsigned long  
    key[BITS_TO_LONGS(KEY_CNT)];
```

```
    int (*open)(struct input_dev *dev); void (*close)(struct input_dev  
    *dev);
```

```
    unsigned int users; struct device dev;
```

```
    unsigned int num_vals; unsigned int max_vals; struct input_value  
    *vals;
```

```
    bool devres_managed; };
```

```
#define ABSMAX_ACC_VAL 0x01FF
```

```
#define ABSMIN_ACC_VAL -(ABSMAX_ACC_VAL) [...]
```

```
set_bit(EV_ABS, idev->evbit);
```

```
input_set_abs_params(idev, ABS_X, ABSMIN_ACC_VAL,
```

```
ABSMAX_ACC_VAL, 0, 0); input_set_abs_params(idev, ABS_Y,  
ABSMIN_ACC_VAL, ABSMAX_ACC_VAL, 0, 0);  
input_set_abs_params(idev, ABS_Z, ABSMIN_ACC_VAL,  
ABSMAX_ACC_VAL, 0, 0);
```

- `key` reflects the current state of the device's keys/buttons.
- `open` is a method called when the very first user calls `input_open_device()`. Use this method to prepare the device, such as interrupt request, polling thread start, and so on.
- `close` is called when the very last user calls `input_close_device()`. Here you can stop polling (which consumes lot of resource).
- `users` stores the number of users (input handlers) that opened this device. It is used by `input_open_device()` and `input_close_device()` to make sure that `dev->open()` is only called when the first user opens the device and `dev->close()` is called when the very last user closes the device.
- `dev` is the struct device associated with this device (for device model).
- `num_vals` is the number of values queued in the current frame.
- `max_vals` is the maximum number of values queued in a frame.
- `vals` is the array of values queued in the current frame.
- `devres_managed` indicates that devices are managed with `devres` framework and needs not be explicitly unregistered or freed.

```
struct input_dev *input_allocate_device(void) struct input_dev  
*devm_input_allocate_device(struct device *dev)
```

```
void input_free_device(struct input_dev *dev) static void  
devm_input_device_unregister(struct device *dev, void *res)
```

```
int input_register_device(struct input_dev *dev) void  
input_unregister_device(struct input_dev *dev)
```

```
struct input_dev *idev;
```

```
int error;
```

```
idev = input_allocate_device();
```

```
if (!idev)
```

```
    return -ENOMEM;
```

```
idev->name = BMA150_DRIVER;
```

```
idev->phys = BMA150_DRIVER "/input0"; idev->id.bustype =  
BUS_I2C;
```

```
idev->dev.parent = &client->dev;
```

```
set_bit(EV_ABS, idev->evbit);
```

```
input_set_abs_params(idev, ABS_X, ABSMIN_ACC_VAL,  
ABSMAX_ACC_VAL, 0, 0); input_set_abs_params(idev, ABS_Y,  
ABSMIN_ACC_VAL, ABSMAX_ACC_VAL, 0, 0);  
input_set_abs_params(idev, ABS_Z, ABSMIN_ACC_VAL,
```



```
ABSMAX_ACC_VAL, 0, 0);  
  
error = input_register_device(idev);  
  
if (error) {  
    input_free_device(idev); return error;  
}
```

```
error = request_threaded_irq(client->irq, NULL, my_irq_thread,  
IRQF_TRIGGER_RISING | IRQF_ONESHOT, BMA150_DRIVER,  
NULL); if (error) {  
    dev_err(&client->dev, "irq request failed %d, error %d\n", client-  
>irq, error); input_unregister_device(bma150->input); goto  
err_free_mem; }
```

```

struct input_polled_dev {

    void *private;


    void (*open)(struct input_polled_dev *dev); void (*close)(struct
input_polled_dev *dev); void (*poll)(struct input_polled_dev *dev);
    unsigned int poll_interval; /* msec */

    unsigned int poll_interval_max; /* msec */

    unsigned int poll_interval_min; /* msec */


    struct input_dev *input;

    bool devres_managed;

};


struct input_polled_dev *devm_input_allocate_polled_device(struct
device *dev) struct input_polled_dev
*input_allocate_polled_device(void) void
input_free_polled_device(struct input_polled_dev *dev)

int input_register_polled_device(struct input_polled_dev *dev) void
input_unregister_polled_device(struct input_polled_dev *dev)

static int button_probe(struct platform_device *pdev) {

    struct my_struct *ms;

```

```
struct input_dev *input_dev; int retval;
```

```
ms = devm_kzalloc(&pdev->dev, sizeof(*ms), GFP_KERNEL); if  
(!ms)
```

```
return -ENOMEM;
```

```
ms->poll_dev = input_allocate_polled_device(); if (!ms->poll_dev)  
{
```

```
kfree(ms);
```

```
return -ENOMEM;
```

```
}
```

```
/* This gpio is not mapped to IRQ */
```

```
ms->reset_btn_desc = gpiod_get(dev, "reset", GPIOD_IN);
```

```
ms->poll_dev->private = ms ; ms->poll_dev->poll = my_btn_poll;  
ms->poll_dev->poll_interval = 200; /* Poll every 200ms */
```

```
ms->poll_dev->open = my_btn_open; /* consist */
```

```
input_dev = ms->poll_dev->input; input_dev->name = "System  
Reset Btn";
```

```

/* The gpio belong to an expander sitting on I2C */

input_dev->id.bustype = BUS_I2C; input_dev->dev.parent =
&pdev->dev;

/* Declare the events generated by this driver */

set_bit(EV_KEY, input_dev->evbit); set_bit(BTN_0, input_dev-
>keybit); /* buttons */


retval = input_register_polled_device(mcp->poll_dev); if (retval) {

dev_err(&pdev->dev, "Failed to register input device\n");
input_free_polled_device(ms->poll_dev); kfree(ms);

}

return retval;

}

struct my_struct {

struct gpio_desc *reset_btn_desc; struct input_polled_dev
*poll_dev; }

static void my_btn_open(struct input_polled_dev *poll_dev) {

struct my_struct *ms = poll_dev->private; dev_dbg(&ms->poll_dev-
>input->dev, "reset open()\n"); }

```

The open method is used to prepare resources needed by the device. We do not really need this method for this example.

Generating and reporting an input event

Device allocation and registration are essential, but they are not the main goal of an input device driver, which is designed to report even to the input core. Depending on the type of event your device can support, the kernel provides appropriate APIs to report them to the core.

Given an `EV_XXX` capable device, the corresponding report function would be `input_report_xxx()`. The following table shows a mapping between the most important event types and their report functions:

Event type	Report function	Code example
EV_KEY	<code>input_report_key()</code>	<pre>input_report_key(poll_dev->input, BTN_0, gpio_get_value(ms->reset_btn_desc) & 1);</pre>
EV_REL	<code>input_report_rel()</code>	<pre>input_report_rel(nunchuk->input, REL_X, (nunchuk->report.joy_x - 128)/10);</pre>
EV_ABS	<code>input_report_abs()</code>	<pre>input_report_abs(bma150->input, ABS_X, x_value); input_report_abs(bma150->input, ABS_Y, y_value); input_report_abs(bma150->input, ABS_Z, z_value);</pre>

Their respective prototypes are as follows: `void input_report_abs(struct input_dev *dev, unsigned int code, int value)` `void input_report_key(struct input_dev *dev, unsigned int code, int value)` `void input_report_rel(struct input_dev *dev, unsigned int code, int value)`

The list of available report functions can be found in `include/linux/input.h` in the kernel source file. They all have the same skeleton:

- `dev` is the input device responsible for the event.
- `code` represents the event code, for example, `REL_X` or `KEY_BACKSPACE`. The complete list is in `include/linux/input-event-codes.h`.
- `value` is the value the event carries. For `EV_REL` event type, it carries the relative change. For a `EV_ABS` (joysticks and so on.) event type, it contains an absolute new value. For `EV_KEY` event type, it should be set to `0` for key release, `1` for key press, and `2` for auto repeat.

After all changes have been reported, the driver should call `input_sync()` on the input device, in order to indicate that this event is complete. The input subsystem will collect these into a single packet and send it through `/dev/input/event<X>`, which is the character device representing our `struct input_dev` on the system, and where `<X>` is the interface number assigned to the driver by the input core: `void input_sync(struct input_dev *dev)`

Let us see an example, which is an excerpt of the `bma150` digital acceleration sensors drivers in `drivers/input/misc/bma150.c`: `static void threaded_report_xyz(struct bma150_data *bma150) { u8 data[BMA150_XYZ_DATA_SIZE]; s16 x, y, z; s32 ret; ret = i2c_smbus_read_i2c_block_data(bma150->client, BMA150_ACC_X_LSB_REG, BMA150_XYZ_DATA_SIZE, data); if (ret != BMA150_XYZ_DATA_SIZE) return; x = ((0xc0 & data[0]) >> 6) | (data[1] << 2); y = ((0xc0 & data[2]) >> 6) | (data[3] << 2); z = ((0xc0 & data[4]) >> 6) | (data[5] << 2); /* sign extension */ x = (s16) (x << 6) >> 6; y = (s16) (y << 6) >> 6; z = (s16) (z << 6) >> 6; input_report_abs(bma150->input, ABS_X, x); input_report_abs(bma150->input, ABS_Y, y); input_report_abs(bma150->input, ABS_Z, z); /* Indicate this event is complete */ input_sync(bma150->input); }`

In the preceding sample, `input_sync()` tells the core to consider the three reports as the same event. It makes sense since the position has three axes (X, Y, Z) and we do not want X, Y, or Z to be reported separately.

The best place to report the event is inside the `poll` function for a polled device, or inside the IRQ routine (threaded part or not) for an IRQ enabled device. If you perform some operations that may sleep, you should process your report inside

the threaded part of the IRQ handled: static void my_btn_poll(struct input_polled_dev *poll_dev) { struct my_struct *ms = poll_dev->private; struct i2c_client *client = mcp->client; input_report_key(poll_dev->input, BTN_0, gpiod_get_value(ms->reset_btn_desc) & 1); input_sync(poll_dev->input); }

```
struct input_event {  
    struct timeval time; __u16 type;  
    __u16 code;  
    __s32 value;  
}  
  
#include <unistd.h>  
  
#include <fcntl.h>  
  
#include <stdio.h>  
  
#include <stdlib.h>  
  
#include <linux/input.h>  
  
#include <sys/select.h>  
  
#define INPUT_DEVICE "/dev/input/event1"  
  
int main(int argc, char **argv)  
{  
    int fd;  
  
    struct input_event event; ssize_t bytesRead;
```



```
int ret;

fd_set readfds;

fd = open(INPUT_DEVICE, O_RDONLY); /* Let's open our input
device */

if(fd < 0){

    fprintf(stderr, "Error opening %s for reading", INPUT_DEVICE);
    exit(EXIT_FAILURE); }


while(1){

    /* Wait on fd for input */

    FD_ZERO(&readfds); FD_SET(fd, &readfds);

    ret = select(fd + 1, &readfds, NULL, NULL, NULL); if (ret == -1)
    {

        fprintf(stderr, "select call on %s: an error occurred",
INPUT_DEVICE); break;

    }

    else if (!ret) { /* If we have decided to use timeout */

        fprintf(stderr, "select on %s: TIMEOUT", INPUT_DEVICE);
        break;

    }

}
```

```

/* File descriptor is now ready */

if (FD_ISSET(fd, &readfds)) {

    bytesRead = read(fd, &event, sizeof(struct input_event));
    if(bytesRead == -1) /* Process read input error*/

        if(bytesRead != sizeof(struct input_event)) /* Read value is not an
input even */


/*

* We could have done a switch/case if we had * many codes to look
for */

    if(event.code == BTN_0) {

        /* it concerns our button */

        if(event.value == 0){

            /* Process Release */

            [...]

        }

        else if(event.value == 1){

            /* Process KeyPress */

            [...]

```

```
}
```

```
}
```

```
}
```

```
}
```

```
close(fd);
```

```
return EXIT_SUCCESS; }
```

```
if(client->irq > 0){  
    /* Use generic input device */  
  
} else {  
  
    /* Use polled device */  
  
}
```

To see how to manage such devices from the user space, please refer to the example provided within the source of the book.

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

#include <linux/module.h>

#include <linux/slab.h>

#include <linux/of.h> /* For DT*/

#include <linux/platform_device.h> /* For platform devices */

#include <linux/gpio/consumer.h> /* For GPIO Descriptor interface
*/

#include <linux/input.h>

#include <linux/input-polldev.h>


struct poll_btn_data {

    struct gpio_desc *btn_gpiod; struct input_polled_dev *poll_dev; };


static void polled_btn_open(struct input_polled_dev *poll_dev) {

    /* struct poll_btn_data *priv = poll_dev->private; */

    pr_info("polled device opened()\n"); }


static void polled_btn_close(struct input_polled_dev *poll_dev) {
```

```

/* struct poll_btn_data *priv = poll_dev->private; */

pr_info("polled device closed()\n"); }

static void polled_btn_poll(struct input_polled_dev *poll_dev) {

    struct poll_btn_data *priv = poll_dev->private;

    input_report_key(poll_dev->input, BTN_0, gpiod_get_value(priv-
>btn_gpiod) & 1); input_sync(poll_dev->input); }

static const struct of_device_id btn_dt_ids[] = {

    { .compatible = "packt,input-polled-button", }, { /* sentinel */ }

};

static int polled_btn_probe(struct platform_device *pdev) {

    struct poll_btn_data *priv; struct input_polled_dev *poll_dev; struct
input_dev *input_dev; int ret;

    priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);
    if (!priv)

        return -ENOMEM;

```

```

poll_dev = input_allocate_polled_device(); if (!poll_dev){
    devm_kfree(&pdev->dev, priv); return -ENOMEM;
}

/* We assume this GPIO is active high */

priv->btn_gpiod = gpiod_get(&pdev->dev, "button", GPIOD_IN);

poll_dev->private = priv; poll_dev->poll_interval = 200; /* Poll
every 200ms */

poll_dev->poll = polled_btn_poll; poll_dev->open =
polled_btn_open; poll_dev->close = polled_btn_close; priv->poll_dev
= poll_dev;

input_dev = poll_dev->input; input_dev->name = "Packt input
polled Btn"; input_dev->dev.parent = &pdev->dev;

/* Declare the events generated by this driver */

set_bit(EV_KEY, input_dev->evbit); set_bit(BTN_0, input_dev-
>keybit); /* buttons */

ret = input_register_polled_device(priv->poll_dev); if (ret) {

    pr_err("Failed to register input polled device\n");
    input_free_polled_device(poll_dev); devm_kfree(&pdev->dev, priv);
    return ret;
}

```

```
}
```

```
platform_set_drvdata(pdev, priv); return 0;
```

```
}
```

```
static int polled_btn_remove(struct platform_device *pdev) {
```

```
    struct poll_btn_data *priv = platform_get_drvdata(pdev);  
    input_unregister_polled_device(priv->poll_dev);  
    input_free_polled_device(priv->poll_dev); gpiod_put(priv->  
    btn_gpiod); return 0;
```

```
}
```

```
static struct platform_driver mypdrv = {
```

```
    .probe = polled_btn_probe, .remove = polled_btn_remove, .driver =  
    {
```

```
        .name = "input-polled-button", .of_match_table =  
        of_match_ptr(btn_dt_ids), .owner = THIS_MODULE,
```

```
    },
```

```
};
```

```
module_platform_driver(mypdrv);
```



```
MODULE_LICENSE("GPL");

MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
MODULE_DESCRIPTION("Polled input device");

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/slab.h>

#include <linux/of.h> /* For DT*/

#include <linux/platform_device.h> /* For platform devices */

#include <linux/gpio/consumer.h> /* For GPIO Descriptor interface
*/

#include <linux/input.h>

#include <linux/interrupt.h>

struct btn_data {

    struct gpio_desc *btn_gpiod; struct input_dev *i_dev; struct
platform_device *pdev; int irq;

};

static int btn_open(struct input_dev *i_dev)
```

```
{  
  
    pr_info("input device opened()\n"); return 0;  
  
}
```

```
static void btn_close(struct input_dev *i_dev) {  
  
    pr_info("input device closed()\n"); }  
  

```

```
static irqreturn_t packt_btn_interrupt(int irq, void *dev_id) {  
  
    struct btn_data *priv = dev_id;  
  
    input_report_key(priv->i_dev, BTN_0, gpiod_get_value(priv->btn_gpiod) & 1); input_sync(priv->i_dev); return IRQ_HANDLED;  
  
}
```

```
static const struct of_device_id btn_dt_ids[] = {  
  
    { .compatible = "packt,input-button", }, { /* sentinel */ }  
  
};
```

```
static int btn_probe(struct platform_device *pdev) {
```

```
    struct btn_data *priv; struct gpio_desc *gpiod; struct input_dev  
    *i_dev; int ret;
```

```
    priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);  
    if (!priv)
```

```
        return -ENOMEM;
```

```
    i_dev = input_allocate_device(); if (!i_dev)
```

```
        return -ENOMEM;
```

```
    i_dev->open = btn_open; i_dev->close = btn_close; i_dev->name =  
    "Packt Btn"; i_dev->dev.parent = &pdev->dev; priv->i_dev = i_dev;  
    priv->pdev = pdev;
```

```
    /* Declare the events generated by this driver */
```

```
    set_bit(EV_KEY, i_dev->evbit); set_bit(BTN_0, i_dev->keybit); /*  
    buttons */
```

```
    /* We assume this GPIO is active high */
```

```
    gpiod = gpiod_get(&pdev->dev, "button", GPIOD_IN); if  
    (IS_ERR(gpiod))
```

```
return -ENODEV;
```

```
priv->irq = gpiod_to_irq(priv->btn_gpiod); priv->btn_gpiod =  
gpiod;
```

```
ret = input_register_device(priv->i_dev); if (ret) {
```

```
pr_err("Failed to register input device\n"); goto err_input;
```

```
}
```

```
ret = request_any_context_irq(priv->irq, packt_btn_interrupt,  
(IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING),  
"packt-input-button", priv); if (ret < 0) {
```

```
dev_err(&pdev->dev, "Unable to acquire interrupt for GPIO  
line\n"); goto err_btn;
```

```
}
```

```
platform_set_drvdata(pdev, priv); return 0;
```

```
err_btn:
```

```
gpiod_put(priv->btn_gpiod); err_input:
```

```
printk("will call input_free_device\n"); input_free_device(i_dev);
```

```
printk("will call devm_kfree\n"); return ret;
```

```
}
```

```
static int btn_remove(struct platform_device *pdev) {
```

```
    struct btn_data *priv; priv = platform_get_drvdata(pdev);  
    input_unregister_device(priv->i_dev); input_free_device(priv->i_dev);  
    free_irq(priv->irq, priv); gpiod_put(priv->btn_gpiod); return  
    0;
```

```
}
```

```
static struct platform_driver mypdrv = {
```

```
    .probe = btn_probe,
```

```
    .remove = btn_remove,
```

```
    .driver = {
```

```
        .name = "input-button", .of_match_table =  
        of_match_ptr(btn_dt_ids), .owner = THIS_MODULE,
```

```
    },
```

```
};
```

```
module_platform_driver(mypdrv);
```

```
MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");  
MODULE_DESCRIPTION("Input device (IRQ based)");
```

```
<strong># udevadm info /dev/input/event0</strong><br/><strong>P:  
/devices/platform/input-button.0/input/input0/event0</strong><br/>  
<strong>N: input/event0</strong><br/><strong>S: input/by-  
path/platform-input-button.0-event</strong><br/><strong>E:  
DEVLINKS=/dev/input/by-path/platform-input-button.0-  
event</strong><br/><strong>E:  
DEVNAME=/dev/input/event0</strong><br/><strong>E:  
DEVPATH=/devices/platform/input-  
button.0/input/input0/event0</strong><br/><strong>E:  
ID_INPUT=1</strong><br/><strong>E: ID_PATH=platform-input-  
button.0</strong><br/><strong>E: ID_PATH_TAG=platform-input-  
button_0</strong><br/><strong>E: MAJOR=13</strong><br/>  
<strong>E: MINOR=64</strong><br/><strong>E:  
SUBSYSTEM=input</strong><br/><strong>E:  
USEC_INITIALIZED=74842430</strong>
```

```
<strong># evtest /dev/input/event0</strong><br/><strong>input  
device opened()</strong><br/><strong>Input driver version is  
1.0.1</strong><br/><strong>Input device ID: bus 0x0 vendor 0x0  
product 0x0 version 0x0</strong><br/><strong>Input device name:  
"Packt Btn"</strong><br/><strong>Supported events:</strong><br/>  
<strong>Event type 0 (EV_SYN)</strong><br/><strong>Event type 1  
(EV_KEY)</strong><br/><strong>Event code 256 (BTN_0)  
</strong>
```

```
<strong>$ cat /proc/interrupts | grep packt</strong><br/>  
<strong>160: 0 0 0 0 gpio-mxc 0 packt-input-button</strong>
```

```
<strong>$ cat /sys/kernel/debug/gpio | grep button</strong><br/>
```

gpio-193 (button-gpio) in hi
\$ cat /sys/kernel/debug/gpio | grep button
gpio-193 (button-gpio) in lo

Summary

This chapter described the whole input framework, and highlighted the difference between polled and interrupt driven input devices. By the end of this chapter, you have the necessary knowledge to write a driver for any input driver, whatever its type, and whatever input event it supports. The user space interface was discussed too, with a sample provided. The next chapter discusses another important framework, the RTC, which is the key element of time management in PC as well as embedded devices.

RTC Drivers

Real Time Clock (RTC) are devices used to track absolute time in nonvolatile memory, which may be internal to the processor, or externally connected through the I2C or SPI bus.

One may use an RTC to do the following:

- Read and set the absolute clock, and generate interrupts during clock updates
- Generate periodic interrupts
- Set alarms

RTCs and the system clock have different purposes. The former is a hardware clock that maintains absolute time and date in a nonvolatile manner, whereas the latter is a software clock maintained by the kernel and used to implement the `gettimeofday(2)` and `time(2)` system calls, as well as setting timestamps on files, and so on. The system clock reports seconds and microseconds from a start point, defined to be the POSIX epoch: 1970-01-01 00:00:00 +0000 (UTC).

In this chapter, we will cover the following topics:

- Introducing RTC framework API
- Describing such driver's architecture, along with a dummy driver example
- Dealing with alarms
- Managing RTC devices from user space, either through the sysfs interface, or using the `hwclock` tool

RTC framework data structures

There are three main data structures used by the RTC framework on Linux systems. They are `struct rtc_time`, `struct rtc_device`, and `struct rtc_class_ops` structures. The former is an opaque structure that represents a given date and time; the second structure represents the physical RTC device; and the last one represents a set of operations exposed by the driver and used by the RTC core to read/update a device's date/time/alarm.

The only header needed to pull RTC functions from within your driver is :

```
| #include <linux/rtc.h>
```

The same file contains all of the three structures enumerated in the preceding section:

```
| struct rtc_time {  
|     int tm_sec; /* seconds after the minute */  
|     int tm_min; /* minutes after the hour - [0, 59] */  
|     int tm_hour; /* hours since midnight - [0, 23] */  
|     int tm_mday; /* day of the month - [1, 31] */  
|     int tm_mon; /* months since January - [0, 11] */  
|     int tm_year; /* years since 1900 */  
|     int tm_wday; /* days since Sunday - [0, 6] */  
|     int tm_yday; /* days since January 1 - [0, 365] */  
|     int tm_isdst; /* Daylight saving time flag */  
| };
```

This structure is similar to the `struct tm` in `<time.h>`, used to pass time. The next structure is `struct rtc_device`, which represent the chip in the kernel:

```
| struct rtc_device {  
|     struct device dev;  
|     struct module *owner;  
  
|     int id;  
|     char name[RTC_DEVICE_NAME_SIZE];  
  
|     const struct rtc_class_ops *ops;  
|     struct mutex ops_lock;  
  
|     struct cdev char_dev;  
|     unsigned long flags;  
  
|     unsigned long irq_data;  
|     spinlock_t irq_lock;  
|     wait_queue_head_t irq_queue;
```

```

    struct rtc_task *irq_task;
    spinlock_t irq_task_lock;
    int irq_freq;
    int max_user_freq;

    struct work_struct irqwork;
};

```

The following are the meanings of the elements of the structure:

- **dev:** This is the device structure.
- **owner:** This is the module that owns this RTC device. Using `THIS_MODULE` will be enough.
- **id:** This is the global index given to the RTC device by the kernel
/dev/rtc<id>.
- **name:** This is the name given to the RTC device.
- **ops:** This is a set of operations (like read/set time/alarm) exposed by this RTC device to be managed by the core or from user space.
- **ops_lock:** This is a mutex used internally by the kernel to protect ops functions call.
- **cdev:** This is the char device associated to this RTC, /dev/rtc<id>.

The next important structure is `struct rtc_class_ops`, which is a set of functions used as callback to perform standard and limited on the RTC device. It is the communication interface between top-layer and bottom-layer RTC drivers:

```

struct rtc_class_ops {
    int (*open)(struct device *);
    void (*release)(struct device *);
    int (*ioctl)(struct device *, unsigned int, unsigned long);
    int (*read_time)(struct device *, struct rtc_time *);
    int (*set_time)(struct device *, struct rtc_time *);
    int (*read_alarm)(struct device *, struct rtc_wkalrm *);
    int (*set_alarm)(struct device *, struct rtc_wkalrm *);
    int (*read_callback)(struct device *, int data);
    int (*alarm_irq_enable)(struct device *, unsigned int enabled);
};

```

All of the hooks in the preceding code are given a `struct device` structure as parameter, which is the same as the one embedded in the `struct rtc_device` structure. This means that from within these hooks, one can access the RTC device itself at any given time, using the `to_rtc_device()` macro, which is built on top of the `container_of()` macro.

```

#define to_rtc_device(d) container_of(d, struct rtc_device, dev)

```

The `open()`, `release()`, and `read_callback()` hooks are internally called by the kernel when the `open()`, `close()`, or `read()` functions are called on the device from user space.

`read_time()` is a driver function that reads the time from the device and fills the `struct rtc_time` output argument. This function should return `0` on success, or else the negative error code.

`set_time()` is a driver function that updates the device's time according to the `struct rtc_time` structure given as the input parameter. Return parameter's remarks are the same as the `read_time` function.

If your device supports an alarm feature, `read_alarm()` and `set_alarm()` should be provided by the driver to read/set the alarm on the device. The `struct rtc_wkalrm` will be described later in the chapter. `alarm_irq_enable()` should be provided too, to enable the alarm.

RTC API

An RTC device is represented in the kernel as an instance of the `struct rtc_device` structure. Unlike other kernel framework devices registrations (where the device is given as parameter to the registering function), the RTC device is built by the core, and registered first before the `rtc_device` structure gets returned to the driver. The device is built and registered with the kernel using the `rtc_device_register()` function:

```
struct rtc_device *rtc_device_register(const char *name,  
                                     struct device *dev,  
                                     const struct rtc_class_ops *ops,  
                                     struct module *owner)
```

One can see the meaning of each parameter of the functions, as follows:

- `name`: This is your RTC device name. It could be the chip's name, for example: `ds1343`.
- `dev`: This is the parent device, used for device model purposes. For chips sitting on I2C or SPI buses, for example, `dev` could be set with `spi_device.dev`, or `i2c_client.dev`.
- `ops`: This is your RTC ops, filled according to the features the RTC has, or those your driver can support.
- `owner`: This is the module to which this RTC device belongs. In most cases, `THIS_MODULE` is enough.

The registration should be performed in the `probe` function, and obviously, one can use the resource-managed version of this function:

```
struct rtc_device *devm_rtc_device_register(struct device *dev,  
                                           const char *name,  
                                           const struct rtc_class_ops *ops,  
                                           struct module *owner)
```

Both functions return a pointer on a `struct rtc_device` structure built by the kernel on success, or a pointer error on which you should use `IS_ERR` and `PTR_ERR` macros.

Associated reverse operations are `rtc_device_unregister()` and `devm_rtc_device_unregister()`:

```
void rtc_device_unregister(struct rtc_device *rtc)
void devm_rtc_device_unregister(struct device *dev,
                               struct rtc_device *rtc)
```

```

int rtc_valid_tm(struct rtc_time *tm);

static int foo_rtc_read_time(struct device *dev, struct rtc_time *tm) {

    struct foo_regs regs;

    int error;


    error = foo_device_read(dev, &regs, 0, sizeof(regs)); if (error)

    return error;


    tm->tm_sec = bcd2bin(regs.seconds); tm->tm_min =
bcd2bin(regs.minutes); tm->tm_hour = bcd2bin(regs.cent_hours); tm-
>tm_mday = bcd2bin(regs.date);

    /*

    * This device returns weekdays from 1 to 7

    * But rtc_time.wday expect days from 0 to 6.

    * So we need to subtract 1 to the value returned by the chip */

    tm->tm_wday = bcd2bin(regs.day) - 1;

    /*

    * This device returns months from 1 to 12

    * But rtc_time.tm_month expect a months 0 to 11.

```

```

    * So we need to subtract 1 to the value returned by the chip */
tm->tm_mon = bcd2bin(regs.month) - 1;

/*

    * This device's Epoch is 2000.

    * But rtc_time.tm_year expect years from Epoch 1900.

    * So we need to add 100 to the value returned by the chip */
tm->tm_year = bcd2bin(regs.years) + 100;

return rtc_valid_tm(tm);
}

#include <linux/bcd.h>

static int foo_rtc_set_time(struct device *dev, struct rtc_time *tm) {

    regs.seconds = bin2bcd(tm->tm_sec); regs.minutes = bin2bcd(tm-
>tm_min); regs.cent_hours = bin2bcd(tm->tm_hour);

    /*

    * This device expects week days from 1 to 7

    * But rtc_time.wday contains week days from 0 to 6.

    * So we need to add 1 to the value given by rtc_time.wday */
regs.day = bin2bcd(tm->tm_wday + 1); regs.date = bin2bcd(tm-

```



```

>tm_mday);

/*
 * This device expects months from 1 to 12
 * But rtc_time.tm_mon contains months from 0 to 11.
 * So we need to add 1 to the value given by rtc_time.tm_mon */
regs.month = bin2bcd(tm->tm_mon + 1);

/*
 * This device expects year since Epoch 2000
 * But rtc_time.tm_year contains year since Epoch 1900.
 * We can just extract the year of the century with the * rest of the
division by 100.
 */

regs.cent_hours |= BQ32K_CENT;

regs.years = bin2bcd(tm->tm_year % 100);

return write_into_device(dev, &regs, 0, sizeof(regs)); }

```

RTC's epoch differs from the POSIX epoch, which is only used for the system clock. If the year according to the RTC's epoch and the year register is less than 1970, it is assumed to be 100 years later, that is, between 2000 and 2069.

```
#include <linux/platform_device.h> #include <linux/module.h>
```

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

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

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

```
#include <linux rtc.h>
```

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

```
static int fake_rtc_read_time(struct device *dev, struct rtc_time *tm) {
```

```
    /*
```

```
    * One can update "tm" with fake values and then call */
```

```
    return rtc_valid_tm(tm); }
```

```
static int fake_rtc_set_time(struct device *dev, struct rtc_time *tm) {
```

```
    return 0;
```

```
}
```

```
static const struct rtc_class_ops fake_rtc_ops = {
```

```
    .read_time = fake_rtc_read_time, .set_time = fake_rtc_set_time };
```

```
static const struct of_device_id rtc_dt_ids[] = {  
    { .compatible = "packt,rtc-fake", }, { /* sentinel */ }  
};
```

```
static int fake_rtc_probe(struct platform_device *pdev) {  
    struct rtc_device *rtc; rtc = rtc_device_register(pdev->name,  
&pdev->dev, &fake_rtc_ops, THIS_MODULE);  
  
    if (IS_ERR(rtc)) return PTR_ERR(rtc);  
  
    platform_set_drvdata(pdev, rtc); pr_info("Fake RTC module  
loaded\n");  
  
    return 0;  
}
```

```
static int fake_rtc_remove(struct platform_device *pdev) {  
    rtc_device_unregister(platform_get_drvdata(pdev)); return 0;  
}
```

```
static struct platform_driver fake_rtc_drv = {
```

```
.probe = fake_rtc_probe, .remove = fake_rtc_remove, .driver = {  
  
    .name = KBUILD_MODNAME, .owner = THIS_MODULE,  
    .of_match_table = of_match_ptr(rtc_dt_ids), },  
  
};  
  
module_platform_driver(fake_rtc_drv);  
  
MODULE_LICENSE("GPL");  
  
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");  
MODULE_DESCRIPTION("Fake RTC driver description");
```

```

struct rtc_wkalrm {

unsigned char enabled; /* 0 = alarm disabled, 1 = enabled */

unsigned char pending; /* 0 = alarm not pending, 1 = pending */

struct rtc_time time; /* time the alarm is set to */

};

void rtc_update_irq(struct rtc_device *rtc, unsigned long num,
unsigned long events)

/* RTC interrupt flags */

#define RTC_IRQF 0x80 /* Any of the following is active */

#define RTC_PF 0x40 /* Periodic interrupt */

#define RTC_AF 0x20 /* Alarm interrupt */

#define RTC_UF 0x10 /* Update interrupt for 1Hz RTC */

static irqreturn_t foo_rtc_alarm_irq(int irq, void *data) {

    struct foo_rtc_struct *foo_device = data; dev_info(foo_device -
>dev, "%s:irq(%d)\n", __func__, irq); rtc_update_irq(foo_device -
>rtc_dev, 1, RTC_IRQF | RTC_AF);

    return IRQ_HANDLED; }

int device_init_wakeup(struct device *dev, bool enable) int
dev_pm_set_wake_irq(struct device *dev, int irq)

static const struct rtc_class_ops foo_rtc_ops = {

```

```
.read_time = foo_rtc_read_time, .set_time = foo_rtc_set_time,  
.read_alarm = foo_rtc_read_alarm, .set_alarm = foo_rtc_set_alarm,  
.alarm_irq_enable = foo_rtc_alarm_irq_enable, .ioctl = foo_rtc_ioctl,  
};
```

```
static int foo_spi_probe(struct spi_device *spi) {
```

```
    int ret;
```

```
    /* initialise and configure the RTC chip */
```

```
    [...]
```

```
    foo_rtc->rtc_dev =
```

```
    devm_rtc_device_register(&spi->dev, "foo-rtc", &foo_rtc_ops,  
    THIS_MODULE);
```

```
    if (IS_ERR(foo_rtc->rtc_dev)) {
```

```
        dev_err(&spi->dev, "unable to register foo rtc\n"); return  
        PTR_ERR(priv->rtc); }
```

```
    foo_rtc->irq = spi->irq;
```

```
    if (foo_rtc->irq >= 0) {
```

```
        ret = devm_request_threaded_irq(&spi->dev, spi->irq, NULL,
```

```
foo_rtc_alarm_irq, IRQF_ONESHOT, "foo-rtc", priv); if (ret) {  
  
    foo_rtc->irq = -1; dev_err(&spi->dev, "unable to request irq for rtc  
foo-rtc\n"); } else {  
  
    device_init_wakeup(&spi->dev, true); dev_pm_set_wake_irq(&spi-  
>dev, spi->irq); }  
  
}  
  
return 0;  
  
}
```

RTCs and user space

On Linux systems, there are two kernel options one needs to care about in order to properly manage RTCs from user space. These are `CONFIG_RTC_HCTOSYS` and `CONFIG_RTC_HCTOSYS_DEVICE`.

`CONFIG_RTC_HCTOSYS` includes the code file `drivers/rtc/hctosys.c` in kernel build process, which sets system time from the RTC on startup and resume. Once this option is enabled, the system time will be set using the value read from the specified RTC device. RTC devices should be specified in

```
CONFIG_RTC_HCTOSYS_DEVICE: CONFIG_RTC_HCTOSYS=y
CONFIG_RTC_HCTOSYS_DEVICE="rtc0"
```

In the preceding example, we tell the kernel to set the system time from the RTC, and we specify that the RTC to use is `rtc0`.

The sysfs interface

The kernel code responsible for instantiating RTC attributes in sysfs is defined in `drivers/rtc/rtc-sysfs.c`, in the kernel source tree. Once registered, an RTC device will create a `rtc<id>` directory under `/sys/class/rtc`. That directory contains a set of read-only attributes, among which the most important are:

- `date`: This file prints the current date of the RTC interface:

```
$ cat /sys/class/rtc/rtc0/date  
2017-08-28
```

- `time`: This prints the current time of this RTC:

```
$ cat /sys/class/rtc/rtc0/time  
14:54:20
```

- `hctosys`: This attribute indicates whether the RTC device is the one specified in `CONFIG_RTC_HCTOSYS_DEVICE`, meaning that this RTC is used to set system time on startup and resume. Read `1` as true, and `0` as false:

```
$ cat /sys/class/rtc/rtc0/hctosys  
  
1
```

- `dev`: This attribute shows the device's major and minor. Read as `major:minor`:

```
$ cat /sys/class/rtc/rtc0/dev  
  
251:0
```

- `since_epoch`: This attribute will print the number of seconds elapsed since the UNIX epoch (since January 1st 1970):

```
$ cat /sys/class/rtc/rtc0/since_epoch  
1503931738
```

The hwclock utility

Hardware clock (hwclock) is a tool used to access RTC devices. The `man hwclock` command will probably be much more meaningful than everything discussed in this section. That said, let us write some commands, to set hwclock RTC from the system clock:

```
$ sudo ntpd -q      # make sure system clock is set from network time
$ sudo hwclock --systohc  # set rtc from the system clock
$ sudo hwclock --show      # check rtc was set
Sat May 17 17:36:50 2017  -0.671045 seconds
```

The preceding example assumes the host has a network connection on which it can access an NTP server. It is also possible to set the system time manually:

```
$ sudo date -s '2017-08-28 17:14:00' '+%s' #set system clock manually
$ sudo hwclock --systohc #synchronize rtc chip on system time
```

If not given as argument, `hwclock` assumes the RTC device file is `/dev/rtc`, which is actually a symbolic link to the real RTC device:

```
$ ls -l /dev/rtc
lrwxrwxrwx 1 root root 4 août 27 17:50 /dev/rtc -> rtc0
```

Summary

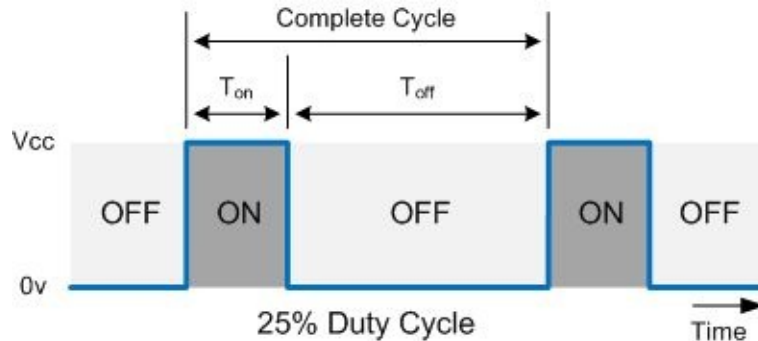
This chapter introduced you to the RTC framework and its API. Its reduced set of functions and data structures make it the most lightweight framework, and easy to master. Using skills described in this chapter, you will be able to develop a driver for most of the existing RTC chips, and even go further and handle such devices from the user space, easily setting up date and time, as well as alarms. The next chapter, PWM drivers, has nothing common with this one, but is a must-know for embedded engineers.

PWM Drivers

Pulse Wide Modulation (PWM) operates like a switch that constantly cycles on and off. It is a hardware feature used to control servomotors, for voltage regulation, and so on. The most well-known applications of PWM are:

- Motor speed control
- Light dimming
- Voltage regulation

Now, let us introduce PWM with a simple following figure:



The preceding figure describes a complete PWM cycle, introducing some terms we need to clarify prior to getting deeper into the kernel PWM framework:

- T_{on} : This is the duration during which the signal is high.
- T_{off} : This is the duration during which the signal is low.
- **Period**: This is the duration of a complete PWM cycle. It represents the sum of T_{on} and T_{off} of the PWM signal.
- **Duty cycle**: It is represented as a percentage of the time signal that remains on during the period of the PWM signal.

Different formulas are detailed as follows:

- PWM period: $T_{on} + T_{off}$
- Duty cycle: $D = \frac{T_{on}}{T_{on} + T_{off}} \times 100 = \frac{T_{on}}{Period} \times 100$



You can find details about PWM at https://en.wikipedia.org/wiki/Pulse-width_



modulation.

The Linux PWM framework has two interfaces:

1. **Controller interface:** The one that exposes the PWM line. It is the PWM chip, that is, the producer.
2. **Consumer interface:** The device consuming PWM lines exposed by the controller. Drivers of such devices use helper functions exported by the controller by means of a generic PWM framework.

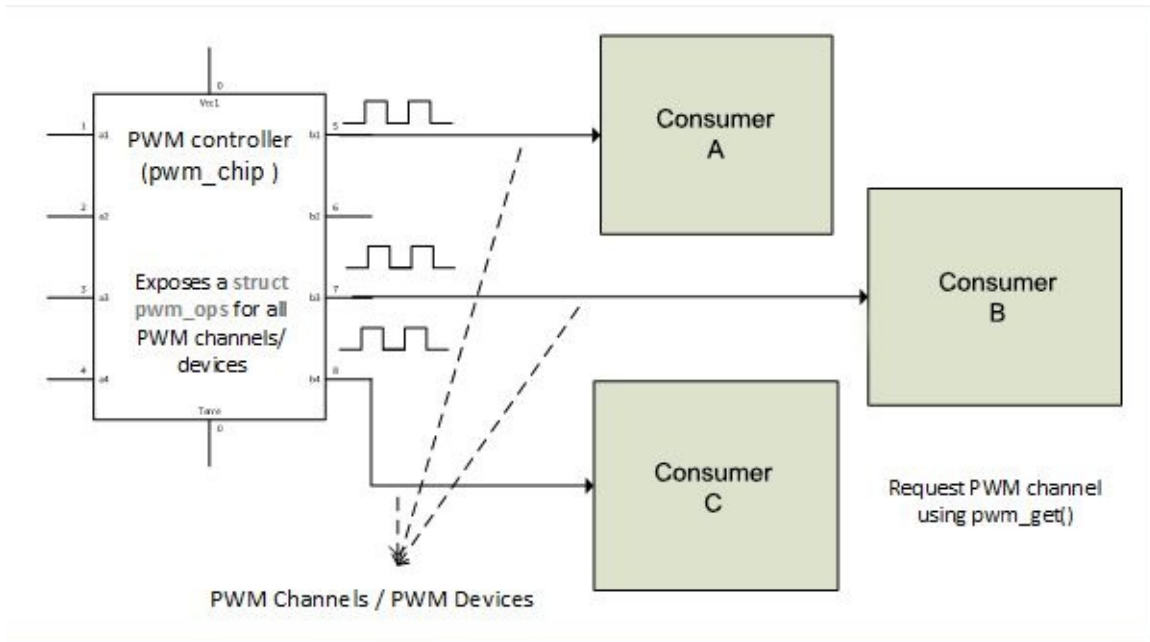
Either the consumer or producer interface depends on the following header file:
`#include <linux/pwm.h>`

In this chapter, we will deal with:

- PWM driver architecture and data structures, for both controller and consumer, along with a dummy driver
- Instantiating PWM devices and controllers in the device tree
- Requesting and consuming PWM devices
- Using PWM from user space through sysfs interface

PWM controller driver

As you need `struct gpio_chip` when writing GPIO-controller drivers and `struct irq_chip` when writing IRQ-controller drivers, a PWM controller is represented in the kernel as an instance of `struct pwm_chip` structure.



PWM controller and devices

```
struct pwm_chip {
    struct device *dev;
    const struct pwm_ops *ops;
    int base;
    unsigned int npwm;

    struct pwm_device *pwms;
    struct pwm_device * (*of_xlate)(struct pwm_chip *pc,
                                    const struct of_phandle_args *args);
    unsigned int of_pwm_n_cells;
    bool can_sleep;
};
```

The following is the meaning of each elements in the structure:

- `dev`: This represents the device associated with this chip.
- `ops`: This is a data structure providing callback functions this chip exposes to consumer drivers.

- **base:** This is the number of the first PWM controlled by this chip. If `chip->base < 0` then, the kernel will dynamically assign a base.
- **can_sleep:** This should be set to `true` by the chip driver if `.config()`, `.enable()`, or `.disable()` operations of the ops field may sleep.
- **npwm:** This is the number of PWM channels (devices) this chip provide.
- **pwms:** This is an array of PWM devices of this chip, allocated by the framework, to consumer drivers.
- **of_xlate:** This is an optional callback to request a PWM device given a DT PWM specifier. If not defined, it will be set to `of_pwm_simple_xlate` by the PWM core, which will force `of_pwm_n_cells` to 2 as well.
- **of_pwm_n_cells:** This is the number of cells expected in the DT for a PWM specifier.

PWM controller/chip adding and removal rely on two basic functions, `pwmchip_add()` and `pwmchip_remove()`. Each function should be given a filled in `struct pwm_chip` structure as an argument. Their respective prototypes are as follows: `int pwmchip_add(struct pwm_chip *chip)` `int pwmchip_remove(struct pwm_chip *chip)`

Unlike other framework removal functions that do not have return values, `pwmchip_remove()` has a return value. It returns 0 on success, or `-EBUSY` if the chip has a PWM line still in use (still requested).

Each PWM driver must implement some hooks through the `struct pwm_ops` field, which is used by the PWM core or the consumer interface in order to configure and make full use of its PWM channels. Some of them are optional.

```
struct pwm_ops {
    int (*request)(struct pwm_chip *chip, struct pwm_device *pwm);
    void (*free)(struct pwm_chip *chip, struct pwm_device *pwm);
    int (*config)(struct pwm_chip *chip, struct pwm_device *pwm,
                  int duty_ns, int period_ns);
    int (*set_polarity)(struct pwm_chip *chip, struct pwm_device *pwm,
                       enum pwm_polarity polarity);
    int (*enable)(struct pwm_chip *chip, struct pwm_device *pwm);
    void (*disable)(struct pwm_chip *chip, struct pwm_device *pwm);
    void (*get_state)(struct pwm_chip *chip, struct pwm_device *pwm,
                     struct pwm_state *state); /* since kernel v4.7 */
    struct module *owner;
};
```

Let us see what each element in the structure means:

- `request`: This is an optional hook that, if provided, is executed during a PWM channel request.
- `free`: This is the same as `request`, ran during PWM freeing.
- `config`: This is the PWM configuration hook. It configures duty cycles and period length for this PWM.
- `set_polarity`: This hook configures the polarity of this PWM.
- `Enable`: This enables the PWM line, starting output toggling.
- `Disable`: This disables the PWM line, stopping output toggling.
- `Apply`: This atomically applies a new PWM config. The state argument should be adjusted with the real hardware config.
- `get_state`: This returns the current PWM state. This function is only called once per PWM device when the PWM chip is registered.
- `owner`: This is the module that owns this chip, usually `THIS_MODULE`.

In the `probe` function of the PWM controller driver, it is good practice to retrieve DT resources, initialize hardware, fill a `struct pwm_chip` and its `struct pwm_ops`, and then, add the PWM chip with the `pwmchip_add` function.

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

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

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

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

```
struct fake_chip {
```

```
    struct pwm_chip chip; int foo;
```

```
    int bar;
```

```
    /* put the client structure here (SPI/I2C) */
```

```
};
```

```
static inline struct fake_chip *to_fake_chip(struct pwm_chip *chip) {
```

```
    return container_of(chip, struct fake_chip, chip); }
```

```
static int fake_pwm_request(struct pwm_chip *chip, struct  
pwm_device *pwm) {
```

```
    /*
```

* One may need to do some initialization when a PWM channel * of the controller is requested. This should be done here.

*

* One may do something like * prepare_pwm_device(struct
pwm_chip *chip, pwm->hwpwm); */

return 0;

}

static int fake_pwm_config(struct pwm_chip *chip, struct
pwm_device *pwm, int duty_ns, int period_ns) {

/*

* In this function, one ne can do something like: * struct fake_chip
*priv = to_fake_chip(chip); *

* return send_command_to_set_config(priv, * duty_ns, period_ns);
*/

return 0;

}

static int fake_pwm_enable(struct pwm_chip *chip, struct

```

pwm_device *pwm) {

    /*

    * In this function, one ne can do something like: * struct fake_chip
    *priv = to_fake_chip(chip); *

    * return foo_chip_set_pwm_enable(priv, pwm->hwpwm, true); */

    pr_info("Somebody enabled PWM device number %d of this chip",
pwm->hwpwm);

    return 0;

}

```

```

static void fake_pwm_disable(struct pwm_chip *chip, struct
pwm_device *pwm) {

    /*

    * In this function, one ne can do something like: * struct fake_chip
    *priv = to_fake_chip(chip); *

    * return foo_chip_set_pwm_enable(priv, pwm->hwpwm, false); */

    pr_info("Somebody disabled PWM device number %d of this chip",
pwm->hwpwm);

```

```
}
```

```
static const struct pwm_ops fake_pwm_ops = {  
    .request = fake_pwm_request, .config = fake_pwm_config, .enable  
= fake_pwm_enable, .disable = fake_pwm_disable, .owner =  
THIS_MODULE, };
```

```
static int fake_pwm_probe(struct platform_device *pdev) {  
    struct fake_chip *priv;  
  
    priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);  
if (!priv)  
  
    return -ENOMEM;  
  
    priv->chip.ops = &fake_pwm_ops; priv->chip.dev = &pdev->dev;  
priv->chip.base = -1; /* Dynamic base */  
  
    priv->chip.npwm = 3; /* 3 channel controller */  
  
    platform_set_drvdata(pdev, priv); return pwmchip_add(&priv-  
>chip); }
```

```

static int fake_pwm_remove(struct platform_device *pdev) {

    struct fake_chip *priv = platform_get_drvdata(pdev); return
    pwmchip_remove(&priv->chip); }


static const struct of_device_id fake_pwm_dt_ids[] = {

    { .compatible = "packt,fake-pwm", }, { }

};

MODULE_DEVICE_TABLE(of, fake_pwm_dt_ids);


static struct platform_driver fake_pwm_driver = {

    .driver = {

        .name = KBUILD_MODNAME, .owner = THIS_MODULE,

        .of_match_table = of_match_ptr(fake_pwm_dt_ids), },

    .probe = fake_pwm_probe, .remove = fake_pwm_remove, };

module_platform_driver(fake_pwm_driver);


MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
MODULE_DESCRIPTION("Fake pwm driver");

MODULE_LICENSE("GPL");

```

```
pwm3: pwm@02088000 {
```

```
    #pwm-cells = <2>; compatible = "fsl,imx6q-pwm", "fsl,imx27-  
pwm"; reg = <0x02088000 0x4000>; interrupts = <0 85  
IRQ_TYPE_LEVEL_HIGH>; clocks = <&clks  
IMX6QDL_CLK_IPG>, <&clks IMX6QDL_CLK_PWM3>; clock-  
names = "ipg", "per"; status = "disabled"; };
```

```
fake_pwm: pwm@0 {
```

```
    #pwm-cells = <2>; compatible = "packt,fake-pwm"; /*
```

```
    * Our driver does not use resource * neither mem, IRQ, nor Clock)  
    */
```

```
};
```

```

struct pwm_device {

    const char *label; unsigned long flags; unsigned int hwpwm;
    unsigned int pwm;

    struct pwm_chip *chip; void *chip_data;

    unsigned int period; /* in nanoseconds */

    unsigned int duty_cycle; /* in nanoseconds */

    enum pwm_polarity polarity; };

struct pwm_device {

    const char *label; unsigned long flags; unsigned int hwpwm;
    unsigned int pwm;

    struct pwm_chip *chip; void *chip_data;

    struct pwm_args args; struct pwm_state state; };

struct pwm_device *pwm_get(struct device *dev, const char *con_id)
void pwm_put(struct pwm_device *pwm)

int pwm_config(struct pwm_device *pwm, int duty_ns, int
period_ns);

int pwm_enable(struct pwm_device *pwm)

void pwm_disable(struct pwm_device *pwm)

```



```
static void pwm_led_drive(struct pwm_device *pwm, struct
private_data *priv) {

    /* Configure the PWM, applying a period and duty cycle */

    pwm_config(pwm, priv->duty, priv->pwm_period);

    /* Start toggling */

    pwm_enable(pchip->pwmd);

    [...] /* Do some work */

    /* And then stop toggling*/

    pwm_disable(pchip->pwmd); }
```

PWM clients binding

PWM devices can be assigned to the consumer from:

- Device tree
- ACPI
- Static lookup tables, in board `init` file.

This book will only deal with DT binding, as it is the recommended method. When binding a PWM consumer (client) to its driver, you need to provide the phandle of the controller to which it is linked.

It is recommended you give the name `pwms` to PWM properties; since PWM devices are named resources, you can provide an optional property `pwm-names`, containing a list of strings to name each of the PWM devices listed in the `pwms` property. In case no `pwm-names` property is given, the name of the user node will be used as fallback.

Drivers for devices that use more than a single PWM device can use the `pwm-names` property to map the name of the PWM device requested by the `pwm_get()` call to an index into the list given by the `pwms` property.

The following example describes a PWM-based backlight device, which is an excerpt from the kernel documentation on PWM device binding (see *Documentation/devicetree/bindings/pwm/pwm.txt*):

```
pwm: pwm {
    #pwm-cells = <2>;
};

[...]

b1: backlight {
    pwms = <&pwm 0 5000000>;
    pwm-names = "backlight";
};
```

The PWM-specifier typically encodes the chip-relative PWM number and the PWM period in nanoseconds. With the line as follows:

```
| pwms = <&pwm 0 5000000>;
```

0 corresponds to the PWM index relative to the controller, and 50000000 represents the period in nanoseconds. Note that in the preceding example, specifying the `pwm-names` is redundant because the name `backlight` would be used as a fallback anyway. Therefore, the driver would have to call:

```
static int my_consumer_probe(struct platform_device *pdev)
{
    struct pwm_device *pwm;

    pwm = pwm_get(&pdev->dev, "backlight");
    if (IS_ERR(pwm)) {
        pr_info("unable to request PWM, trying legacy API\n");
        /*
         * Some drivers use the legacy API as fallback, in order
         * to request a PWM ID, global to the system
         * pwm = pwm_request(global_pwm_id, "pwm beeper");
         */
    }

    [...]
    return 0;
}
```



The PWM-specifier typically encodes the chip-relative PWM number and the PWM period in nanoseconds.

** # echo 1 >**

/sys/class/pwm/pwmchip<pwmchipnr>/pwm<pwmnr>/enable

2. Set PWM period:

echo <value in nanoseconds> > /sys/class/pwm/

3. Set PWM duty cycle: The value of the duty cycle must be less than the value of PWM period:

echo <value in nanoseconds> > /sys/class/pwm/pwmch

4. Disable PWM:

echo 0 > /sys/class/pwm/pwmchip<pwmchipnr>/pwm<pw

The complete PWM framework API and sysfs description is available in the *Documentation/pwm.txt* file, in the kernel source tree.

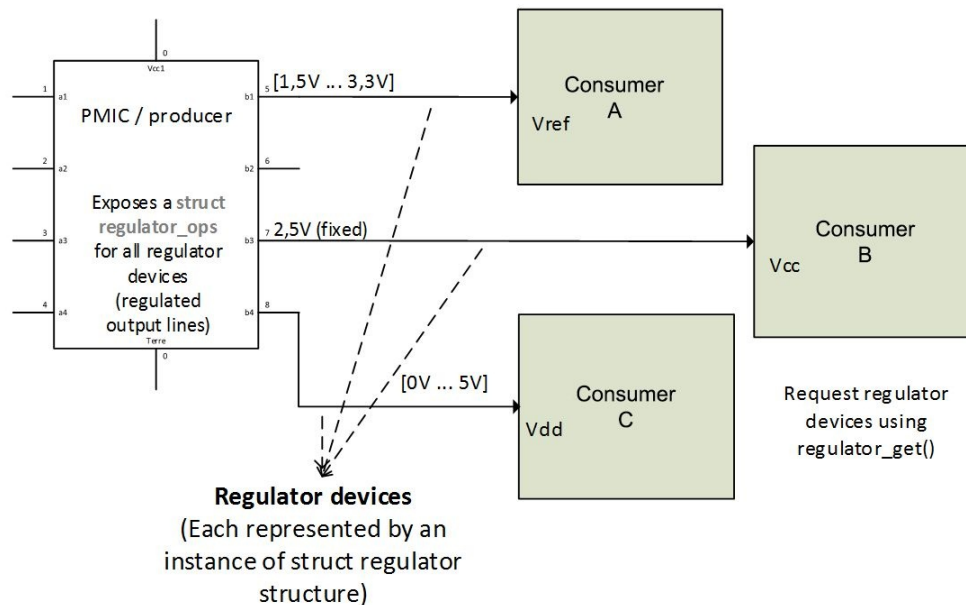
Summary

By the end of this chapter, you are armed for any PWM controller, whether it is memory mapped, or externally sitting on a bus. The API described in this chapter will be sufficient to write as well as to enhance a controller driver as a consumer device driver. If you are not comfortable with the PWM kernel side yet, you can fully use the user space sysfs interface. That said, in the next chapter, we will discuss about regulators, which sometimes are driven by PWM. So, please hold on, we are almost done.

Regulator Framework

A regulator is an electronic device that supplies power to other devices. Devices powered by regulators are called consumers. One said they consume power provided by regulators. Most regulators can enable and disable their output and some can also control their output voltage or current. The driver should expose those capabilities to consumers by means of specific functions and data structures, which we will discuss in this chapter.

The chip that physically provides regulators is called a **Power Management Integrated Circuit (PMIC)**:



The Linux regulator framework has been designed to interface and control voltage and current regulators. It is divided into four separate interfaces as follows:

- A regulator drivers interface for regulator PMIC drivers. The structure of this interface can be found in `include/linux/regulator/driver.h`.
- A consumer interface for device drivers.
- A machine interface for board configuration.
- A sysfs interface for user space.

In this chapter, we will cover the following topics:

- Introducing the PMIC/producer driver interface, driver methods and data structures
- A case study with ISL6271A MIC driver, as well as a dummy regulator for testing purpose
- A regulator consumer interface along with its API
- Regulator (producer/consumer) binding in DT

```
#include <linux/platform_device.h> #include  
<linux/regulator/driver.h> #include <linux/regulator/of_regulator.h>
```


Driver data structures

We will start with a short walkthrough of data structures used by the regulator framework. Only the producer interface is described in this section.

```

struct regulator_desc {

    const char *name; const char *of_match;

    int id; unsigned n_voltages; const struct regulator_ops *ops; int irq;
    enum regulator_type type; struct module *owner;

    unsigned int min_uV; unsigned int uV_step; };

```

Let us omit some fields for simplicity reasons. Full structure definition is available in `include/linux/regulator/driver.h`:

- `name` holds the name of the regulator.
- `of_match` holds the name used to identify the regulator in DT.
- `id` is a numerical identifier for the regulator.
- `owner` represents the module providing the regulator. Set this field to `THIS_MODULE`.
- `type` indicates if the regulator is a voltage regulator or a current regulator. It can either be `REGULATOR_VOLTAGE` or `REGULATOR_CURRENT`. Any other value will result in a regulator registering failure.
- `n_voltages` indicates the number of selectors available for this regulator. It represents the numerical value that the regulator can output. For fixed output voltage, `n_voltages` should be set to 1.
- `min_uV` indicates the minimum voltage value this regulator can provide. It is the voltage given by the lowest selector.
- `uV_step` represents the voltage increase with each selector.
- `ops` represents the regulator operations table. It is a structure pointing to a set of operation callbacks that the regulator can support. This field is discussed later.
- `irq` is the interrupt number of the regulator.

```
struct regulation_constraints {  
    const char *name;  
  
    /* voltage output range (inclusive) - for voltage control */  
    int min_uV;  
    int max_uV;  
  
    int uV_offset;  
  
    /* current output range (inclusive) - for current control */  
    int min_uA;  
    int max_uA;  
  
    /* valid regulator operating modes for this machine */  
    unsigned int valid_modes_mask;  
  
    /* valid operations for regulator on this machine */  
    unsigned int valid_ops_mask;  
  
    struct regulator_state state_disk; struct regulator_state state_mem;
```

```

struct regulator_state state_standby; suspend_state_t initial_state; /*
suspend state to set at init */

/* mode to set on startup */

unsigned int initial_mode;

/* constraint flags */

unsigned always_on:1; /* regulator never off when system is on */

unsigned boot_on:1; /* bootloader/firmware enabled regulator */

unsigned apply_uV:1; /* apply uV constraint if min == max */

};

```

Let us describe each element in the structure:

- `min_uV`, `min_uA`, `max_uA`, and `max_uV` are the smallest voltage/current values that the consumers may set.
- `uV_offset` is the offset applied to voltages from the consumer to compensate for voltage drops.
- `valid_modes_mask` and `valid_ops_mask` respectively are masks of modes/operations which may be configured/performed by consumers.
- `always_on` should be set if the regulator should never be disabled.
- `boot_on` should be set if the regulator is enabled when the system is initially started. If the regulator is not enabled by the hardware or bootloader then it will be enabled when the constraints are applied.
- `name` is a descriptive name for the constraints used for display purposes.

- `apply_uv` applies the voltage constraint when initializing.
- `input_uv` represents the input voltage for this regulator when it is supplied by another regulator.
- `state_disk`, `state_mem`, and `state_standby` define the state for the regulator when the system is suspended in the disk mode, mem mode, or in standby.
- `initial_state` indicates the suspended state is set by default.
- `initial_mode` is the mode to set at startup.

```
struct regulator_init_data {  
  
    struct regulation_constraints constraints;  
  
    /* optional regulator machine specific init */  
  
    int (*regulator_init)(void *driver_data); void *driver_data; /* core  
does not touch this */  
  
};
```

The following are the meanings of elements in the structure:

- `constraints` represents the regulator constraints
- `regulator_init` is an optional callback invoked at a given moment when the core registers the regulator
- `driver_data` represents the data passed to `regulator_init`

As one can see, the `struct constraints` structure is part of the `init` data. This is explained by the fact that at the initialization of the regulator, its constraint is directly applied to it, far before any consumer can use it.

```

static struct regulator_init_data isl_init_data[] = {

    [0] = {

        .constraints = {

            .name = "Core Buck", .min_uV = 850000, .max_uV = 1600000,
            .valid_modes_mask = REGULATOR_MODE_NORMAL

            | REGULATOR_MODE_STANDBY, .valid_ops_mask =
            REGULATOR_CHANGE_MODE

            | REGULATOR_CHANGE_STATUS, },

        },

    [1] = {

        .constraints = {

            .name = "LDO1", .min_uV = 1100000, .max_uV = 1100000,
            .always_on = true, .valid_modes_mask =
            REGULATOR_MODE_NORMAL

            | REGULATOR_MODE_STANDBY, .valid_ops_mask =
            REGULATOR_CHANGE_MODE

            | REGULATOR_CHANGE_STATUS, },

        },

    [2] = {

        .constraints = {

```

```
.name = "LDO2", .min_uV = 1300000, .max_uV = 1300000,  
.always_on = true, .valid_modes_mask =  
REGULATOR_MODE_NORMAL  
  
| REGULATOR_MODE_STANDBY, .valid_ops_mask =  
REGULATOR_CHANGE_MODE  
  
| REGULATOR_CHANGE_STATUS, },  
  
},  
  
};
```

This method is now depreciated, though it is presented here for your information. The new and recommended approach is the DT, which is described in the next section.


```
struct of_regulator_match {
```

```
    const char *name; void *driver_data; struct regulator_init_data  
    *init_data; struct device_node *of_node; const struct regulator_desc  
    *desc; };
```

```
isl6271a@3c {
```

```
    compatible = "isl6271a"; reg = <0x3c>; interrupts = <0 86 0x4>;
```

```
    /* supposing our regulator is powered by another regulator */
```

```
    in-v1-supply = <&some_reg>; [...]
```

```
    regulators {
```

```
        reg1: core_buck {
```

```
            regulator-name = "Core Buck"; regulator-min-microvolt =  
<850000>; regulator-max-microvolt = <1600000>; };
```

```
        reg2: ldo1 {
```

```
            regulator-name = "LDO1"; regulator-min-microvolt = <1100000>;  
            regulator-max-microvolt = <1100000>; regulator-always-on; };
```

```
        reg3: ldo2 {
```

```
            regulator-name = "LDO2"; regulator-min-microvolt = <1300000>;
```

```
regulator-max-microvolt = <1300000>; regulator-always-on; };
```

```
};
```

```
};
```

Using the kernel helper function of `_regulator_match()`, given the regulators sub-node as the parameter, the function will walk through each regulator device node and build a `struct init_data` structure for each of them. There is an example in the `probe()` function, discussed in the driver methods section.

```
struct regulator_config {
```

```
    struct device *dev; const struct regulator_init_data *init_data; void  
    *driver_data; struct device_node *of_node; };
```

- dev represents the struct device structure the regulator belongs to.
- init_data is the most important field of the structure, since it contains an element holding the regulator constraints (a machine specific structure).
- driver_data holds the regulator's private data.
- of_node is for DT capable drivers. It is the node to parse for DT bindings. It is up to the developer to set this field. It may be NULL also.

```
struct regulator_ops {  
  
    /* enumerate supported voltages */  
  
    int (*list_voltage) (struct regulator_dev *, unsigned selector);  
  
  
    /* get/set regulator voltage */  
  
    int (*set_voltage) (struct regulator_dev *, int min_uV, int max_uV,  
        unsigned *selector);  
  
    int (*map_voltage)(struct regulator_dev *, int min_uV, int  
max_uV);  
  
    int (*set_voltage_sel) (struct regulator_dev *, unsigned selector);  
  
    int (*get_voltage) (struct regulator_dev *); int (*get_voltage_sel)  
(struct regulator_dev *);  
  
    /* get/set regulator current */  
  
    int (*set_current_limit) (struct regulator_dev *, int min_uA, int  
max_uA);  
  
    int (*get_current_limit) (struct regulator_dev *);  
  
    int (*set_input_current_limit) (struct regulator_dev *, int lim_uA);  
  
    int (*set_over_current_protection) (struct regulator_dev *); int  
(*set_active_discharge) (struct regulator_dev *, bool enable);  
}
```

```
/* enable/disable regulator */

int (*enable) (struct regulator_dev *); int (*disable) (struct
regulator_dev *); int (*is_enabled) (struct regulator_dev *);

/* get/set regulator operating mode (defined in consumer.h) */

int (*set_mode) (struct regulator_dev *, unsigned int mode);
unsigned int (*get_mode) (struct regulator_dev *); };

int rdev_get_id(struct regulator_dev *rdev)
```

Driver methods

Driver methods consist of `probe()` and `remove()` functions. Please refer to the preceding data structure if this section seems unclear to you.

```
struct regulator_dev * regulator_register(const struct regulator_desc
*regulator_desc, const struct regulator_config *cfg)
```

This function returns a data type we have not discussed so far: a `struct regulator_dev` object, defined in `include/linux/regulator/driver.h`. That structure represents an instance of a regulator device from the producer side (it is different in the consumer side). Instances of the `struct regulator_dev` structure should not be used directly by anything except the regulator core and notification injection (which should take the mutex and not other direct access). That being said, to keep track of the registered regulator from within the driver, one should hold references for each `regulator_dev` object returned by the registering function.

```
void regulator_unregister(struct regulator_dev *rdev)

static int __devexit isl6271a_remove(struct i2c_client *i2c) {

    struct isl_pmic *pmic = i2c_get_clientdata(i2c); int i;


    for (i = 0; i < 3; i++) regulator_unregister(pmic->rdev[i]);

    kfree(pmic); return 0; }
```



```

struct isl_pmic {

    struct i2c_client *client; struct regulator_dev *rdev[3]; struct mutex
    mtx; };

static int isl6271a_probe(struct i2c_client *i2c, const struct
i2c_device_id *id) {

    struct regulator_config config = { }; struct regulator_init_data
    *init_data =

    dev_get_platdata(&i2c->dev);

    struct isl_pmic *pmic;

    int i, ret;


    struct device *dev = &i2c->dev; struct device_node *np, *parent;

    if (!i2c_check_functionality(i2c->adapter,
I2C_FUNC_SMBUS_BYTE_DATA)) return -EIO;


    pmic = devm_kzalloc(&i2c->dev, sizeof(struct isl_pmic),
GFP_KERNEL); if (!pmic)

    return -ENOMEM;

    /* Get the device (PMIC) node */

    np = of_node_get(dev->of_node); if (!np)

```

```

return -EINVAL;

/* Get 'regulators' subnode */

parent = of_get_child_by_name(np, "regulators"); if (!parent) {
dev_err(dev, "regulators node not found\n"); return -EINVAL; }

/* fill isl6271a_matches array */

ret = of_regulator_match(dev, parent, isl6271a_matches,
ARRAY_SIZE(isl6271a_matches));

of_node_put(parent); if (ret < 0) {
dev_err(dev, "Error parsing regulator init data: %d\n", ret);

return ret;
}

pmic->client = i2c; mutex_init(&pmic->mtx);

for (i = 0; i < 3; i++) {

struct regulator_init_data *init_data; struct regulator_desc *desc; int
val;

if (pdata)

```

```

/* Given as platform data */

config.init_data = pdata->init_data[i]; else

/* Fetched from device tree */

config.init_data = isl6271a_matches[i].init_data;

config.dev = &i2c->dev; config.of_node =
isl6271a_matches[i].of_node; config.ena_gpio = -EINVAL;

/*

* config is passed by reference because the kernel * internally
duplicate it to create its own copy * so that it can override some fields
*/

pmic->rdev[i] = devm_regulator_register(&i2c->dev, &isl_rd[i],
&config); if (IS_ERR(pmic->rdev[i])) {

dev_err(&i2c->dev, "failed to register %s\n", id->name);

return PTR_ERR(pmic->rdev[i]); }

}

i2c_set_clientdata(i2c, pmic); return 0;

}

/* Only the first regulator actually need it */

if (i == 0)

```

```
if(pdata)

config.init_data = init_data; /* pdata */

else

isl6271a_matches[i].init_data; /* DT */
```

else

```
config.init_data = NULL;
```

The preceding driver does not fill every field of the struct `regulator_desc`. It greatly depends on the type of device for which we write a driver. Some drivers leave the whole job to the regulator core, and only provide the chip's register address, which the regulator core needs to work with. Such drivers use **regmap** API, which is a generic I2C and SPI register map library. `drivers/regulator/max8649.c` is an example.

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

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/platform_device.h> /* For platform devices */

#include <linux/interrupt.h> /* For IRQ */

#include <linux/of.h> /* For DT*/

#include <linux/err.h>

#include <linux/regulator/driver.h> #include
<linux/regulator/machine.h>

#define DUMMY_VOLTAGE_MIN 850000

#define DUMMY_VOLTAGE_MAX 1600000

#define DUMMY_VOLTAGE_STEP 50000

struct my_private_data {

    int foo;

    int bar;

    struct mutex lock; };

```

```
static const struct of_device_id regulator_dummy_ids[] = {  
    { .compatible = "packt,regulator-dummy", }, { /* sentinel */ }  
};
```

```
static struct regulator_init_data dummy_initdata[] = {  
    [0] = {  
        .constraints = {  
            .always_on = 0,  
            .min_uV = DUMMY_VOLTAGE_MIN, .max_uV =  
DUMMY_VOLTAGE_MAX, },  
        },  
    [1] = {  
        .constraints = {  
            .always_on = 1,  
        },  
        },  
};
```

```
static int isl6271a_get_voltage_sel(struct regulator_dev *dev) {
```

```
    return 0;

}
```

```
static int isl6271a_set_voltage_sel(struct regulator_dev *dev,
unsigned selector) {

    return 0;

}
```

```
static struct regulator_ops dummy_fixed_ops = {

    .list_voltage = regulator_list_voltage_linear, };


```

```
static struct regulator_ops dummy_core_ops = {

    .get_voltage_sel = isl6271a_get_voltage_sel, .set_voltage_sel =
isl6271a_set_voltage_sel, .list_voltage =
regulator_list_voltage_linear, .map_voltage =
regulator_map_voltage_linear, };


```

```
static const struct regulator_desc dummy_desc[] = {

{

    .name = "Dummy Core", .id = 0,
```



```
.n_voltages = 16, .ops = &dummy_core_ops, .type =  
REGULATOR_VOLTAGE, .owner = THIS_MODULE, .min_uV =  
DUMMY_VOLTAGE_MIN, .uV_step =  
DUMMY_VOLTAGE_STEP, }, {
```

```
.name = "Dummy Fixed", .id = 1,
```

```
.n_voltages = 1,
```

```
.ops = &dummy_fixed_ops, .type = REGULATOR_VOLTAGE,  
.owner = THIS_MODULE, .min_uV = 1300000, },
```

```
};
```

```
static int my_pdrv_probe (struct platform_device *pdev) {
```

```
    struct regulator_config config = { }; config.dev = &pdev->dev;
```

```
    struct regulator_dev *dummy_regulator_rdev[2];
```

```
    int ret, i;
```

```
    for (i = 0; i < 2; i++){
```

```
        config.init_data = &dummy_initdata[i]; dummy_regulator_rdev[i] =  
        \
```

```
        regulator_register(&dummy_desc[i], &config); if  
(IS_ERR(dummy_regulator_rdev)) {
```

```
            ret = PTR_ERR(dummy_regulator_rdev); pr_err("Failed to register  
regulator: %d\n", ret); return ret;
```

```
}
```

```
}
```

```
platform_set_drvdata(pdev, dummy_regulator_rdev); return 0;
```

```
}
```

```
static void my_pdrv_remove(struct platform_device *pdev) {
```

```
    int i;
```

```
    struct regulator_dev *dummy_regulator_rdev = \
```

```
    platform_get_drvdata(pdev); for (i = 0; i < 2; i++)  
    regulator_unregister(&dummy_regulator_rdev[i]); }
```

```
static struct platform_driver mypdrv = {
```

```
    .probe = my_pdrv_probe, .remove = my_pdrv_remove, .driver = {
```

```
    .name = "regulator-dummy", .of_match_table =  
    of_match_ptr(regulator_dummy_ids), .owner = THIS_MODULE, },
```

```
};
```

```
module_platform_driver(mypdrv);
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
```

```
MODULE_LICENSE("GPL");
```

Dummy Core: at 850 mV
Dummy Fixed: 1300 mV

**# ls /sys/class/regulator/
regulator.0 regulator.11
regulator.14 regulator.4 regulator.7
regulator.1 regulator.12 regulator.2 regulator.5
regulator.8
regulator.10
regulator.13 regulator.3 regulator.6 regulator.9**

cd /sys/class/regulator
**# cat
regulator.13/name**
Dummy Core
cat regulator.14/name
Dummy Fixed
**# cat
regulator.14/type**
voltage
cat regulator.14/microvolts
1300000
**# cat
regulator.13/microvolts**
850000

```
#include <linux/regulator/consumer.h>
```

```
/*
```

```
* struct regulator
```

```
*
```

```
* One for each consumer device.
```

```
*/
```

```
struct regulator {
```

```
    struct device *dev; struct list_head list; unsigned int always_on:1;  
    unsigned int bypass:1; int uA_load; int min_uV; int max_uV; char  
    *supply_name; struct device_attribute dev_attr; struct regulator_dev  
    *rdev; struct dentry *debugfs; };
```

```
[...]
```

```
int ret;
```

```
struct regulator *reg;
```

```
const char *supply = "vdd1"; int min_uV, max_uV;
```

```
reg = regulator_get(dev, supply); [...]
```

Regulator device requesting

Prior to gaining access to a regulator, the consumer has to request the kernel by means of the `regulator_get()` function. It is also possible to use the managed version, the `devm_regulator_get()` function:

```
| struct regulator *regulator_get(struct device *dev,  
| const char *id)
```

An example of using this function is:

```
| reg = regulator_get(dev, "Vcc");
```

The consumer passes in its `struct device` pointer and power supply ID. The core will try to find the correct regulator by consulting the DT or a machine-specific lookup table. If we focus only on the device tree, `*id` should match the `<name>` pattern of the regulator supply in the device tree. If the lookup is successful then this call will return a pointer to the `struct regulator` that supplies this consumer.

To release the regulator, the consumer driver should call:

```
| void regulator_put(struct regulator *regulator)
```

Prior to calling this function, the driver should ensure that all `regulator_enable()` calls made on this regulator source are balanced by `regulator_disable()` calls.

More than one regulator can supply a consumer, for example, codec consumers with analog and digital supplies:

```
| digital = regulator_get(dev, "Vcc"); /* digital core */  
| analog = regulator_get(dev, "Avdd"); /* analog */
```

Consumer `probe()` and `remove()` functions are an appropriate place to grab and release regulators.

Controlling the regulator device

Regulator control consists of enabling, disabling, and setting output values for a regulator.

Regulator output enable and disable

A consumer can enable its power supply by calling the following:

```
| int regulator_enable(regulator);
```

This function returns 0 on success. The reverse operation consists of disabling the power supply, by calling this:

```
| int regulator_disable(regulator);
```

To check whether a regulator is already enabled or not, the consumer should call this:

```
| int regulator_is_enabled(regulator);
```

This function returns a value greater than 0 if the regulator is enabled. Since the regulator may be enabled early by the bootloader or shared with another consumer, one can use the `regulator_is_enabled()` function to check the regulator state.

Here is an example,

```
| printk (KERN_INFO "Regulator Enabled = %d\n",  
          regulator_is_enabled(reg));
```



For a shared regulator, `regulator_disable()` will actually disable the regulator only when the enabled reference count is zero. That said, you can force disabling in case of an emergency, for example, by calling `regulator_force_disable()`:

```
| int regulator_force_disable(regulator);
```

Each of the functions that we will discuss in the sections that follows is actually a wrapper around a `regulator_ops` operation. For example, `regulator_set_voltage()` internally calls `regulator_ops.set_voltage` after checking the corresponding mask allowing this operation is set, and so on.

Voltage control and status

For consumers that need to adapt their power supplies according to their operating modes, the kernel provides this: `int regulator_set_voltage(regulator, min_uV, max_uV);`

`min_uV` and `max_uV` are the minimum and maximum acceptable voltages in microvolts.

If called when the regulator is disabled, this function will change the voltage configuration so that the voltage is physically set when the regulator is next enabled. That said, consumers can get the regulator configured voltage output by calling `regulator_get_voltage()`, which will return the configured output voltage whether the regulator is enabled or not: `int regulator_get_voltage(regulator);`

Here is an example,

```
| printk (KERN_INFO "Regulator Voltage = %d\n",  
| regulator_get_voltage(reg));
```


Current limit control and status

What we have discussed in the voltage section also applies here. For example, USB drivers may want to set the limit to 500 mA when supplying power.

Consumers can control their supply current limit by calling: `int regulator_set_current_limit(regulator, min_uA, max_uA);`

`min_uA` and `max_uA` are the minimum and maximum acceptable current limits in microamps.

In the same way, consumers can get the regulator configured to the current limit by calling `regulator_get_current_limit()`, which will return the current limit whether the regulator is enabled or not: `int regulator_get_current_limit(regulator);`

```
int regulator_set_optimum_mode(struct regulator *regulator, int  
load_uA);
```

```
int regulator_set_mode(struct regulator *regulator, unsigned int  
mode); unsigned int regulator_get_mode(struct regulator *regulator);
```

Consumers should use `regulator_set_mode()` on a regulator only when it knows about the regulator and does not share the regulator with other consumers. This is known as **direct mode**.

`regulator_set_optimum_mode()` causes the core to undertake some background work in order to determine what operating mode is best for the requested current. This is called the **indirect mode**.

<name>-supply: phandle to the regulator node

```
twl_reg1: regulator@0 {
```

```
    [...]
```

```
};
```

```
twl_reg2: regulator@1 {
```

```
    [...]
```

```
};
```

```
mmc: mmc@0x0 {
```

```
    [...]
```

```
    vmmc-supply = <&twl_reg1>; vmmcaux-supply = <&twl_reg2>; };
```

```
struct regulator *main_regulator; struct regulator *aux_regulator; int  
ret;
```

```
main_regulator = devm_regulator_get(dev, "vmmc");
```

```
/*
```

```
* It is a good practice to apply the config before * enabling the  
regulator */
```

```
if (!IS_ERR(io_regulator)) {
```

```
    regulator_set_voltage(main_regulator,  
MMC_VOLTAGE_DIGITAL, MMC_VOLTAGE_DIGITAL); ret =  
regulator_enable(io_regulator); }
```

```
[...]
```

```
aux_regulator = devm_regulator_get(dev, "vmmcaux"); [...]
```

Summary

With the wide range of devices that need to be smartly and smoothly supplied, this chapter can be relied on to take care of their power supply management. PMIC devices usually sit on SPI or I2C buses. Having already dealt with these buses in previous chapters, you should be able to write any PMIC driver. Let's now jump to the next chapter, which deals with framebuffer drivers, which is a completely different and no less interesting topic.

Framebuffer Drivers

Video cards always have a certain amount of RAM. This RAM is where the bitmap of image data is buffered for display. From the software point of view, the framebuffer is a character device providing access to this RAM.

That said, a framebuffer driver provides an interface for:

- Display mode setting
- Memory access to the video buffer
- Basic 2D acceleration operations (for example, scrolling)

To provide this interface, the framebuffer driver generally talks to the hardware directly. There are well-known framebuffer drivers such as:

- **intelfb**, which is a framebuffer for various Intel 8xx/9xx compatible graphic devices
- **vesafb**, which is a framebuffer driver that uses the VESA standard interface to talk to the video hardware
- **mxcfb**, the framebuffer driver for i.MX6 chip series

Framebuffer drivers are the simplest form of graphics drivers under Linux, not to confuse them with X.org drivers, which implement advanced features such as 3D acceleration and so on, or Kernel mode setting (KMS) drivers, which expose both framebuffer and GPU functionalities (like X.org drivers do).



*i.MX6 X.org driver is a closed source and called **vivante**.*

Back to our framebuffer drivers, they are very simple API drivers that expose video card functionalities by means of character devices, accessible from the user space through `/dev/fbX` entries. One can find more information on Linux graphical stack in the comprehensive talk *Linux Graphics Demystified* by Martin Fiedler: http://keyj.emphy.de/files/linuxgraphics_en.pdf.

In this chapter, we cover the following topics:

- Framebuffer driver data structures and methods, thus covering the whole driver architecture
- Framebuffer device operations, accelerated and non-accelerated
- Accessing framebuffer from user space

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

```
struct fb_info *framebuffer_alloc(size_t size, struct device *dev)
```

```
void framebuffer_release(struct fb_info *info)
```

```
int register_framebuffer(struct fb_info *fb_info)
```

```
int unregister_framebuffer(struct fb_info *fb_info)
```

Allocation and registering should be done during the device probing, whereas unregistering and deallocation (release) should be done from within the driver's `remove()` function.


```

struct fb_ops {

    /* open/release and usage marking */

    struct module *owner;

    int (*fb_open)(struct fb_info *info, int user); int (*fb_release)(struct
fb_info *info, int user);

    /* For framebuffers with strange nonlinear layouts or that do not *
work with normal memory mapped access */

    ssize_t (*fb_read)(struct fb_info *info, char __user *buf, size_t
count, loff_t *ppos);

    ssize_t (*fb_write)(struct fb_info *info, const char __user *buf,
size_t count, loff_t *ppos);

    /* checks var and eventually tweaks it to something supported, *
DO NOT MODIFY PAR */

    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info
*info);

    /* set the video mode according to info->var */

    int (*fb_set_par)(struct fb_info *info);

    /* set color register */

    int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,
unsigned blue, unsigned transp, struct fb_info *info);

```

```

/* set color registers in batch */

int (*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);

/* blank display */

int (*fb_blank)(int blank_mode, struct fb_info *info);

/* pan display */

int (*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info
*info);

/* Draws a rectangle */

void (*fb_fillrect) (struct fb_info *info, const struct fb_fillrect
*rect); /* Copy data from area to another */

void (*fb_copyarea) (struct fb_info *info, const struct fb_copyarea
*region); /* Draws a image to the display */

void (*fb_imageblit) (struct fb_info *info, const struct fb_image
*image);

/* Draws cursor */

int (*fb_cursor) (struct fb_info *info, struct fb_cursor *cursor);

/* wait for blit idle, optional */

int (*fb_sync)(struct fb_info *info);

/* perform fb specific ioctl (optional) */

int (*fb_ioctl)(struct fb_info *info, unsigned int cmd, unsigned long

```

```
arg);
```

```
/* Handle 32bit compat ioctl (optional) */
```

```
int (*fb_compat_ioctl)(struct fb_info *info, unsigned cmd, unsigned  
long arg);
```

```
/* perform fb specific mmap */
```

```
int (*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);
```

```
/* get capability given var */
```

```
void (*fb_get_caps)(struct fb_info *info, struct fb_blit_caps *caps,  
struct fb_var_screeninfo *var);
```

```
/* teardown any resources to do with this framebuffer */
```

```
void (*fb_destroy)(struct fb_info *info); [...]
```

```
};
```

Different callbacks can be set depending on what functionality one wishes to implement.

In [Chapter 4](#), *Character Device Drivers*, we learned that character devices, by means of `struct file_operations` structure, can export a collection of file operations, which are entry points for file-related system calls such as `open()`, `close()`, `read()`, `write()`, `mmap()`, `ioctl()`, and so on.

That being said, do not confuse `fb_ops` with `file_operations` structure. `fb_ops` offers an abstraction of low-level operations, while `file_operations` is for an upper-level system call interface. The kernel implements framebuffer file operations in `drivers/video/fbdev/core/fbmem.c`, which internally call methods we defined in `fb_ops`. In this manner, one can implement the low-level hardware operations according to the need of the system call interface, namely the `file_operations` structure. For example, when the user `open()` the device, the core's `open` file operation method will perform some core operations, and execute `fb_ops.fb_open()` method if set, same for `release`, `mmap`, and so on.

Framebuffer devices support some `ioctl` commands defined in `include/uapi/linux/fb.h`, that user programs can use to operate on hardware. These commands are all handled by the core's `fops.ioctl` method. For some of those commands, the core's `ioctl` method may internally execute methods defined in `fb_ops` structure.

One may wonder what the `fb_ops.ffb_ioctl` is used for. The framebuffer core executes `fb_ops.fb_ioctl` only when the given `ioctl` command is not known to the kernel. In other words, `fb_ops.fb_ioctl` is executed in the default statement of the framebuffer core's `fops.ioctl` method.

```
static struct fb_ops myfb_ops = {

    .owner = THIS_MODULE, .fb_check_var = myfb_check_var,
    .fb_set_par = myfb_set_par, .fb_setcolreg = myfb_setcolreg,
    .fb_fillrect = cfb_fillrect, /* Those three hooks are */

    .fb_copyarea = cfb_copyarea, /* non accelerated and */

    .fb_imageblit = cfb_imageblit, /* are provided by kernel */

    .fb_blank = myfb_blank, };
```

- Probe: Driver probe function is in charge of initializing the hardware, creating the struct fb_info structure using framebuffer_alloc() function, and register_framebuffer() on it. The following sample assumes the device is memory mapped. Therefore, your nonmemory map can exist, such as screen sitting on SPI buses. In this case, bus specific routines should be used:

```
static int myfb_probe(struct platform_device *pdev)
{
    struct fb_info *info;
    struct resource *res;
    [...]

    dev_info(&pdev->dev, "My framebuffer driver\n");

    /*
     * Query resource, like DMA channels, I/O memory,
     * regulators, and so on.
     */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0)
    if (!res)
        return -ENODEV;
    /* use request_mem_region(), ioremap() and so on */
    [...]
    pwr = regulator_get(&pdev->dev, "lcd");
```

```

    info = framebuffer_alloc(sizeof(
struct my_private_struct), &pdev->dev);
    if (!info)
        return -ENOMEM;

    /* Device init and default info value*/
    [...]
    info->fbops = &myfb_ops;

    /* Clock setup, using devm_clk_get() and so on */
    [...]

    /* DMA setup using dma_alloc_coherent() and so on*/
    [...]

    /* Register with the kernel */
    ret = register_framebuffer(info);

    hardware_enable_controller(my_private_struct);
    return 0;
}

```

- Remove: The remove() function should release whatever was acquired in probe(), and call:

```

static int myfb_remove(struct platform_device *pdev)
{
    /* iounmap() memory and release_mem_region() */
    [...]
    /* Reverse DMA, dma_free_*();*/
    [...]

    hardware_disable_controller(fbi);

    /* first unregister, */
    unregister_framebuffer(info);
    /* and then free the memory */
    framebuffer_release(info);

    return 0;
}

```

}

- Assuming you used the manager version for resource allocations, you'll just need to use `unregister_framebuffer()` and `framebuffer_release()`. Everything else will be done by the kernel.

Detailed fb_ops

Let us describe some of the hooks declared in `fb_ops` structure. That being said, for an idea on writing framebuffer drivers, you can have a look at `drivers/video/fbdev/vfb.c`, which is a simple virtual framebuffer driver in the kernel. One can also have a look at other specific framebuffer drivers, like i.MX6 one, at `drivers/video/fbdev/imxfb.c`, or at the kernel documentation about framebuffer driver API at `Documentation/fb/api.txt`.


```

int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info
*info);

static int myfb_check_var(struct fb_var_screeninfo *var, struct
fb_info *info)

{

    if (var->xres_virtual < var->xres) var->xres_virtual = var->xres;

    if (var->yres_virtual < var->yres) var->yres_virtual = var->yres;

    if ((var->bits_per_pixel != 32) && (var->bits_per_pixel != 24) &&
(var->bits_per_pixel != 16) && (var->bits_per_pixel != 12) && (var-
>bits_per_pixel != 8)) var->bits_per_pixel = 16;

    switch (var->bits_per_pixel) {

case 8:

    /* Adjust red*/

    var->red.length = 3; var->red.offset = 5; var->red.msb_right = 0;

    /*adjust green*/

    var->green.length = 3; var->green.offset = 2; var->green.msb_right
= 0;

    /* adjust blue */

    var->blue.length = 2; var->blue.offset = 0; var->blue.msb_right = 0;

    /* Adjust transparency */

```

```

    var->transp.length = 0; var->transp.offset = 0; var-
>transp.msb_right = 0; break;

    case 16: [...]

    break;

    case 24: [...]

    break;

    case 32: var->red.length = 8; var->red.offset = 16; var-
>red.msb_right = 0;

    var->green.length = 8; var->green.offset = 8; var->green.msb_right
= 0;

    var->blue.length = 8; var->blue.offset = 0; var->blue.msb_right = 0;

    var->transp.length = 8; var->transp.offset = 24; var-
>transp.msb_right = 0; break;

}

/*

* Any other field in *var* can be adjusted * like var->xres, var->yres,
var->bits_per_pixel, * var->pixclock and so on.

*/

return 0; }

```

The preceding code adjusts variable framebuffer properties according to the configuration chosen by user.

```
static int myfb_set_par(struct fb_info *info) {  
    struct fb_var_screeninfo *var = &info->var;  
  
    /* Make some compute or other sanity check */  
  
    [...]  
  
    /*  
     * This function writes value to the hardware, * in the appropriate  
     registers */  
  
    set_controller_vars(var, info);  
  
    return 0; }
```

```

int (*fb_blank)(int blank_mode, struct fb_info *info)

enum {

    /* screen: unblanked, hsync: on, vsync: on */

    FB_BLANK_UNBLANK = VESA_NO_BLANKING,

    /* screen: blanked, hsync: on, vsync: on */

    FB_BLANK_NORMAL = VESA_NO_BLANKING + 1,

    /* screen: blanked, hsync: on, vsync: off */

    FB_BLANK_VSYNC_SUSPEND = VESA_VSYNC_SUSPEND +
1,

    /* screen: blanked, hsync: off, vsync: on */

    FB_BLANK_HSYNC_SUSPEND = VESA_HSYNC_SUSPEND +
1,

    /* screen: blanked, hsync: off, vsync: off */

    FB_BLANK_POWERDOWN = VESA_POWERDOWN + 1

};

static int myfb_blank(int blank_mode, struct fb_info *info) {

    pr_debug("fb_blank: blank=%d\n", blank);

    switch (blank) {

        case FB_BLANK_POWERDOWN: case

```

```
FB_BLANK_VSYNC_SUSPEND: case  
FB_BLANK_HSYNC_SUSPEND: case FB_BLANK_NORMAL:  
myfb_disable_controller(fbi); break;
```

```
    case FB_BLANK_UNBLANK: myfb_enable_controller(fbi);  
break;
```

```
}
```

```
return 0;
```

```
}
```

Blanking operation should disable the controller, stop its clocks and power it down. Unblanking should perform the reverse operations.

Accelerated methods

Users video operations such as blending, stretching, moving bitmaps, or dynamic gradient generation are all heavy-duty tasks. They require graphics acceleration to obtain acceptable performance. One can implement framebuffer accelerated methods using the following fields of `struct fp_ops` structure:

- `.fb_imageblit()`: This method draws an image on the display and is very useful
- `.fb_copyarea()`: This method copies a rectangular area from one screen region to another
- `.fb_fillrect()`: This method fills in an optimized manner a rectangle with pixel lines

Therefore, kernel developers thought of controllers that did not have hardware acceleration, and provided a software-optimized method. This makes acceleration implementation optional, since software fall-back exists. That said, if the framebuffer controller does not provide any acceleration mechanism, one must populate these methods using the kernel generic routines.

These are respectively:

- `cfb_imageblit()`: This is a kernel-provided fallback for `imageblit`. The kernel uses it to output a logo to the screen during boot up.
- `cfb_copyarea()`: This is for area copy operations.
- `cfb_fillrect()`: This is the framebuffer core non-accelerated method to achieve operations of the same name.

Putting it all together

In this section, let us summarize things discussed in the preceding section. In order to write framebuffer driver, one has to:

- Fill a `struct fb_var_screeninfo` structure in order to provide information on framebuffer variable properties. Those properties can be changed by user space.
- Fill a `struct fb_fix_screeninfo` structure, to provide fixed parameters.
- Set up a `struct fb_ops` structure, providing necessary callback functions, which will be used by the framebuffer subsystem in response to user actions.
- Still in the `struct fb_ops` structure, one has to provide accelerated functions callback, if supported by the device.
- Set up a `struct fb_info` structure, feeding it with structures filled in previous steps, and call `register_framebuffer()` on it in order to have it registered with the kernel.

For an idea on writing a simple framebuffer driver, one can have a look at `drivers/video/fbdev/vfb.c`, which is a virtual framebuffer driver in kernel. One can enable this in the kernel by means of the `CONFIG_FB_VIRTUAL` option.


```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#include <fcntl.h>
```

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

```
#include <sys/mman.h>
```

```
#include <sys/ioctl.h>
```

```
#define FBCTL(_fd, _cmd, _arg) \
    if(ioctl(_fd, _cmd, _arg) == -1) { \
        ERROR("ioctl failed"); \
        exit(1); }
```

```
int main()
```

```
{
```

```
    int fd;
```

```
    int x, y, pos;
```

```
    int r, g, b;
```

```
unsigned short color; void *fbmem;
```

```
struct fb_var_screeninfo var_info; struct fb_fix_screeninfo fix_info;
```

```
fd = open(FBVIDEO, O_RDWR); if (tfd == -1 || vfd == -1) {
```

```
exit(-1);
```

```
}
```

```
/* Gather variable screen info (virtual and visible) */
```

```
FBCTL(fd, FBIOGET_VSCREENINFO, &var_info);
```

```
/* Gather fixed screen info */
```

```
FBCTL(fd, FBIOGET_FSCREENINFO, &fix_info);
```

```
printf("***** Frame Buffer Info *****\n"); printf("Visible:  
%d,%d \nvirtual: %d,%d \n line_len %d\n", var_info.xres, this-  
>var_info.yres, var_info.xres_virtual, var_info.yres_virtual,  
fix_info.line_length); printf("dim %d,%d\n\n", var_info.width,  
var_info.height);
```

```
/* Let's mmap frame buffer memory */
```

```
fbmem = mmap(0, v_var.yres_virtual * v_fix.line_length, \
```

```
PROT_WRITE | PROT_READ, \
```

```
MAP_SHARED, fd, 0);
```

```

if (fbmem == MAP_FAILED) {

perror("Video or Text frame bufer mmap failed"); exit(1);

}


/* upper left corner (100,100). The square is 300px width */

for (y = 100; y < 400; y++) {

for (x = 100; x < 400; x++) {

    pos = (x + vinfo.xoffset) * (vinfo.bits_per_pixel / 8) + (y +
vinfo.yoffset) * finfo.line_length;

    /* if 32 bits per pixel */

    if (vinfo.bits_per_pixel == 32) {

/* We prepare some blue color */

*(fbmem + pos) = 100;

/* adding a little green */

*(fbmem + pos + 1) = 15+(x-100)/2;

/* With lot of read */

*(fbmem + pos + 2) = 200-(y-100)/5;

/* And no transparency */

```

```

*(fbmem + pos + 3) = 0; } else { /* This assume 16bpp */

r = 31-(y-100)/16;

g = (x-100)/6;

b = 10;


/* Compute color */

color = r << 11 | g << 5 | b; *((unsigned short int*)(fbmem + pos)) =
color; }

}

}

munmap(fbp, screensize); close(fbfd);

return 0;

}

<strong> # cat /dev/fb0 > my_image </strong>

<strong> # cat my_image > /dev/fb0 </strong>

<strong> # echo 0 > /sys/class/graphics/fb0/blank</strong> <strong>#
echo 1 > /sys/class/graphics/fb0/blank</strong>

```

Summary

The framebuffer drivers are the simplest form of Linux graphics drivers, requiring little implementation work. They heavily abstract hardware. At this stage, you should be able to enhance either an existing driver (with graphical acceleration functions for example), or write a fresh one from scratch. However, it is recommended to rely on an existing driver whose hardware shares as many characteristics as possible with the one you need to write the driver for. Let us jump to the next and last chapter, dealing with network devices.

Network Interface Card Drivers

We all know that networking is inherent to the Linux kernel. Some years ago, Linux was only used for its network performances, but things have changed now; Linux is much more than a server, and runs on billions of embedded devices. Through the years, Linux gained the reputation of being the best network operating system. In spite of all this, Linux cannot do everything. Given the huge variety of Ethernet controllers that exist, Linux has found no other way than to expose an API to developers who need a writing driver for their network device, or who need to perform kernel networking development in a general manner. This API offers a sufficient abstraction layer, allowing for guaranteeing the generosity of the developed code, as well as porting on other architectures. This chapter will simply walk through the part of this API that deals with **Network Interface Card (NIC)** driver development, and discuss its data structures and methods.

In this chapter, we will cover the following topics:

- NIC driver data structure and a walk through its main socket buffer structure
- NIC driver architecture and methods description, as well as packets transmission and reception
- Developing a dummy NIC driver for testing purposes

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

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

```
struct sk_buff {
```

```
    struct sk_buff * next; struct sk_buff * prev; ktime_t tstamp; struct  
    rb_node rbnode; /* used in netem & tcp stack */
```

```
    struct sock * sk; struct net_device * dev; unsigned int len; unsigned  
    int data_len; __u16 mac_len; __u16 hdr_len; unsigned int len;  
    unsigned int data_len; __u16 mac_len; __u16 hdr_len; __u32 priority;  
    dma_cookie_t dma_cookie; sk_buff_data_t tail; sk_buff_data_t end;  
    unsigned char * head; unsigned char * data; unsigned int truesize;  
    atomic_t users; };
```

The following is the meanings of the elements in the structure:

- next and prev : This represents the next and previous buffer in the list.
- sk: This is the socket associated with this packet.
- tstamp: This is the time when the packet arrived/left.
- rbnode: This is an alternative to next/prev represented in a red-black tree.
- dev: This represents the device this packet arrived on/is leaving by. This field is associated with two other fields not listed here. These are input_dev and real_dev. They track devices associated with the packet. Therefore, input_dev always refers to a device the packet is received from.
- len: This is the total number of bytes in the packet. Socket Buffers (SKBs) are composed of a linear data buffer and, optionally, a set of one or more regions called **rooms**. In case there are such rooms, data_len will hold the total number of bytes of the data area.
- mac_len: This holds the length of the MAC header.
- csum: This contains the checksum of the packet.
- Priority: This represents the packet priority in QoS.

- `true_size`: This keeps track of how many bytes of system memory are consumed by a packet, including the memory occupied by the `struct sk_buff` structure itself.
- `users`: This is used for reference counting for the SKB objects.
- `head`: `head`, `data`, `tail` are pointers to different regions (rooms) in the socket buffer.
- `end`: This points to the end of the socket buffer.

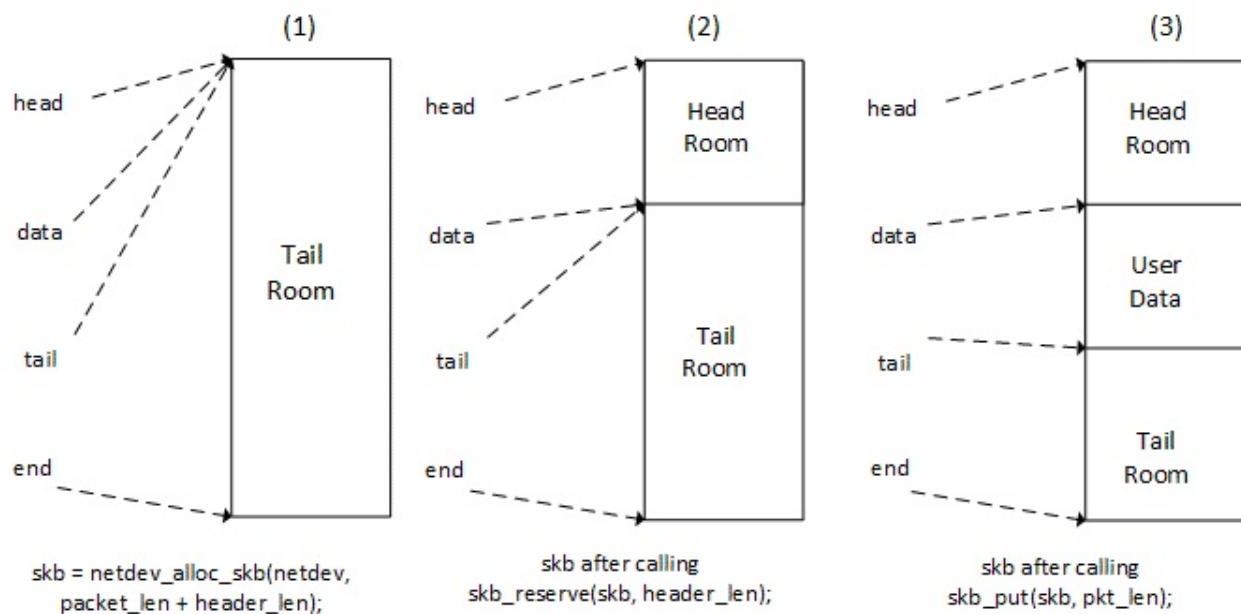
Only a few fields of this structure have been discussed here. A full description is available in `include/linux/skbuff.h`, which is the header file you should include to deal with socket buffers.

Socket buffer allocation

Allocation of a socket buffer is a bit tricky, since it needs at least three different functions:

- First of all, the whole memory allocation should be done using the `netdev_alloc_skb()` function
- Increase and align header room with the `skb_reserve()` function
- Extend the used data area of the buffer (which will contain the packet) using the `skb_put()` function.

Let us have a look at the following figure:



Socket buffers allocation process

1. We allocate a buffer large enough to contain a packet along with the Ethernet header by means of the `netdev_alloc_skb()` function:

```
struct sk_buff *netdev_alloc_skb(struct net_device *dev,  
                                unsigned int length)
```

This function returns `NULL` on failure. Therefore, even if it allocates

memory, `netdev_alloc_skb()` can be called from an atomic context.

Since the Ethernet header is 14 bytes long, it needs to have some alignment so that the CPU does not encounter any performance issues while accessing that part of the buffer. The appropriate name of the `header_len` parameter should be `header_alignment`, since this parameter is used for alignment. The usual value is 2, and it is the reason why the kernel defined a dedicated macro for this purpose, `NET_IP_ALIGN`, in

`include/linux/skbuff.h`:

```
| #define NET_IP_ALIGN 2
```

2. The second step reserves aligned memory for the header by reducing the tail room. The function that does is `skb_reserve()`:

```
| void skb_reserve(struct sk_buff *skb, int len)
```

3. The last step consists of extending the used data area of the buffer to as large as the packet size, by means of the `skb_put()` function. This function returns a pointer to the first byte of the data area:

```
| unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
```

The allocated socket buffer should be forwarded to the kernel-networking layer. This is the last step of the socket buffer's lifecycle. One should use the `netif_rx_ni()` function for that:

```
| int netif_rx_ni(struct sk_buff *skb)
```

We will discuss how to use the preceding steps in the section of this chapter that deals with packet reception.

```

struct net_device {

    char name[IFNAMSIZ]; char *ifalias;

    unsigned long mem_end; unsigned long mem_start; unsigned long
    base_addr; int irq;

    netdev_features_t features; netdev_features_t hw_features;
    netdev_features_t wanted_features; int ifindex;

    struct net_device_stats stats; atomic_long_t rx_dropped;
    atomic_long_t tx_dropped; const struct net_device_ops *netdev_ops;
    const struct ethtool_ops *ethtool_ops; unsigned int flags; unsigned int
    priv_flags; unsigned char link_mode; unsigned char if_port; unsigned
    char dma; unsigned int mtu; unsigned short type; /* Interface address
    info. */

    unsigned char perm_addr[MAX_ADDR_LEN]; unsigned char
    addr_assign_type; unsigned char addr_len; unsigned short
    neigh_priv_len; unsigned short dev_id; unsigned short dev_port;
    unsigned long last_rx; /* Interface address info used in
    eth_type_trans() */

    unsigned char *dev_addr;

    struct device dev; struct phy_device *phydev; };

struct net_device *alloc_etherdev(int sizeof_priv);

void *netdev_priv(const struct net_device *dev)

struct net_device *net_dev;

struct priv_struct *priv_net_struct;

```

```
net_dev = alloc_etherdev(sizeof(struct priv_struct)); my_priv_struct =  
netdev_priv(dev);
```

```
void free_netdev(struct net_device *dev)
```

```
int register_netdev(struct net_device *dev)
```

```

struct net_device_ops {

    int (*ndo_init)(struct net_device *dev); void (*ndo_uninit)(struct
net_device *dev); int (*ndo_open)(struct net_device *dev); int
(*ndo_stop)(struct net_device *dev); netdev_tx_t (*ndo_start_xmit)
(struct sk_buff *skb, struct net_device *dev);

    void (*ndo_change_rx_flags)(struct net_device *dev, int flags);
void (*ndo_set_rx_mode)(struct net_device *dev); int
(*ndo_set_mac_address)(struct net_device *dev, void *addr); int
(*ndo_validate_addr)(struct net_device *dev); int (*ndo_do_ioctl)
(struct net_device *dev, struct ifreq *ifr, int cmd);

    int (*ndo_set_config)(struct net_device *dev, struct ifmap *map);
int (*ndo_change_mtu)(struct net_device *dev, int new_mtu); void
(*ndo_tx_timeout) (struct net_device *dev);

    struct net_device_stats* (*ndo_get_stats)(
        struct net_device *dev);

};

```

Let us see what the meaning of each element in the structure is:

- `int (*ndo_init)(struct net_device *dev)` and `void (*ndo_uninit)(struct net_device *dev);` They are extra initialization/uninitialization functions, respectively executed when the driver calls `register_netdev()`/`unregister_netdev()` in order to register/unregister the network device with the kernel. Most drivers do not provide those functions, since the real job is done by `ndo_open()` and `ndo_stop()` functions.
- `int (*ndo_open)(struct net_device *dev);` Prepares and

opens the interface. The interface is opened whenever `ip` or `ifconfig` utilities activate it. In this method, the driver should request/map/register any system resource it needs (I/O ports, IRQ, DMA, and so on), turn on the hardware, and perform any other setup the device requires.

- `int (*ndo_stop)(struct net_device *dev)`: The kernel executes this function when the interface is brought down (For example, `ifconfig <name> down` and so on). This function should perform reverse operations of what has been done in `ndo_open()`.
- `int (*ndo_start_xmit)(struct sk_buff *skb, struct net_device *dev)`: This method is called whenever the kernel wants to send a packet through this interface.
- `void (*ndo_set_rx_mode)(struct net_device *dev)`: This method is called to change the interface address list filter mode, multicast or promiscuous. It is recommended to provide this function.
- `void (*ndo_tx_timeout)(struct net_device *dev)`: The kernel calls this method when a packet transmission fails to complete within a reasonable period, usually for `dev->watchdog` ticks. The driver should check what happened, handle the problem, and resume packet transmission.
- `struct net_device_stats *(*get_stats)(struct net_device *dev)`: This method returns the device statistic. It is what one can see when `netstat -i` or `ifconfig` is run.

The preceding descriptions miss a lot of fields. The complete structure description is available in the `include/linux/netdevice.h` file.

Actually, only `ndo_start_xmit` is mandatory, but it is a good practice to provide as many helper hooks as your device has features.

```
/*
```

```
* This routine should set everything up new at each open, even *  
registers that should only need to be set once at boot, so that * there is  
non-reboot way to recover if something goes wrong.
```

```
*/
```

```
static int enc28j60_net_open(struct net_device *dev) {
```

```
    struct priv_net_struct *priv = netdev_priv(dev);
```

```
    if (!is_valid_ether_addr(dev->dev_addr)) {
```

```
        [...] /* Maybe print a debug message ? */
```

```
        return -EADDRNOTAVAIL; }
```

```
/*
```

```
* Reset the hardware here and take it out of low * power mode
```

```
*/
```

```
    my_netdev_lowpower(priv, false);
```

```
    if (!my_netdev_hw_init(priv)) {
```

```
        [...] /* handle hardware reset failure */
```

```
        return -EINVAL;
```

```
    }
```



```
/* Update the MAC address (in case user has changed it) * The new  
address is stored in netdev->dev_addr field */
```

```
set_hw_macaddr_registers(netdev, MAC_REGADDR_START,  
netdev->addr_len, netdev->dev_addr);
```

```
/* Enable interrupts */
```

```
my_netdev_hw_enable(priv);
```

```
/* We are now ready to accept transmit requests from * the  
queueing layer of the networking.
```

```
*/
```

```
netif_start_queue(dev);
```

```
return 0;
```

```
}
```

```
/* The inverse routine to net_open(). */
```

```
static int enc28j60_net_close(struct net_device *dev) {
```

```
    struct priv_net_struct *priv = netdev_priv(dev);
```

```
    my_netdev_hw_disable(priv); my_netdev_lowpower(priv, true);
```

```
/**
```

```
 * netif_stop_queue - stop transmitted packets *
```

```
 * Stop upper layers calling the device ndo_start_xmit routine.
```

* Used for flow control when transmit resources are unavailable.

*/

netif_stop_queue(dev);

return 0;

}

netif_stop_queue() simply does the reverse of netif_start_queue(), telling the kernel to stop calling the device ndo_start_xmit routine. We can't handle transmit request anymore.

Packet handling

Packet handling consists of transmission and reception of packets. This is the main task of any network interface driver. Transmission refers only to sending outgoing frames, whereas reception refers to frames coming in.

There are two ways to drive networking data exchange: by polling or by interrupt. Polling, which is a kind of timer-driven interrupt, consists of a kernel continuously checking at given intervals for any change from the device. On the other hand, interrupt mode consists of the kernel doing nothing, listening to an IRQ line, and waiting for the device to notify a change, by means of the IRQ. Interrupt-driven data exchange can increase system overhead during time of high traffic. That is why some drivers mix the two methods. The part of the kernel that allows mixing of the two methods is called **New API (NAPI)**, which consists of using polling during times of high traffic and using interrupt IRQ-driven management when the traffic becomes normal. New drivers should use NAPI if the hardware can support it. However, NAPI is not discussed in this chapter, which will focus on the interrupt-driven method.

```
/*
```

```
* RX handler
```

```
* This function is called in the work responsible of packet * reception  
(bottom half) handler. We use work because access to * our device  
(which sit on a SPI bus) may sleep */
```

```
static int my_rx_interrupt(struct net_device *ndev) {
```

```
    struct priv_net_struct *priv = netdev_priv(ndev); int pk_counter,  
ret;
```

```
    /* Let's get the number of packet our device received */
```

```
    pk_counter = my_device_reg_read(priv, REG_PKT_CNT);
```

```
    if (pk_counter > priv->max_pk_counter) {
```

```
        /* update statistics */
```

```
        priv->max_pk_counter = pk_counter; }
```

```
    ret = pk_counter;
```

```
    /* set receive buffer start */
```

```
    priv->next_pk_ptr = KNOWN_START_REGISTER; while  
(pk_counter-- > 0) /*
```

```
* By calling this internal helper function in a "while"
```

* loop, packets get extracted one by one from the device * and forwarder to the network layer.

*/

my_hw_rx(ndev);

return ret;

}

/*

* Hardware receive function.

* Read the buffer memory, update the FIFO pointer to * free the buffer.

* This function decrements the packet counter.

*/

static void my_hw_rx(struct net_device *ndev) {

struct priv_net_struct *priv = netdev_priv(ndev); struct sk_buff *skb = NULL; u16 erxdpt, next_packet, rxstat; u8 rsv[RSV_SIZE];

int packet_len;

packet_len = my_device_read_current_packet_size(); /* Can't cross boundaries */

```

if ((priv->next_pk_ptr > RXEND_INIT)) {

/* packet address corrupted: reset RX logic */

[...]

/* Update RX errors stats */

ndev->stats.rx_errors++; return;

}

/* Read next packet pointer and rx status vector * This is device-
specific */

my_device_reg_read(priv, priv->next_pk_ptr, sizeof(rsv), rsv);

/* Check for errors in the device RX status reg, * and update error
stats accordingly */

if(an_error_is_detected_in_device_status_registers()) /* Depending
on the error, * stats.rx_errors++;

* ndev->stats.rx_crc_errors++; * ndev->stats.rx_frame_errors++; *
ndev->stats.rx_over_errors++; */

} else {

skb = netdev_alloc_skb(ndev, len + NET_IP_ALIGN); if (!skb) {

ndev->stats.rx_dropped++; } else {

skb_reserve(skb, NET_IP_ALIGN); /*

* copy the packet from the device' receive buffer * to the socket

```

buffer data memory.

* Remember `skb_put()` return a pointer to the * beginning of data region.

*/

```
my_netdev_mem_read(priv, rx_packet_start(priv->next_pk_ptr),  
len, skb_put(skb, len));
```

```
/* Set the packet's protocol ID */
```

```
skb->protocol = eth_type_trans(skb, ndev); /* update RX statistics  
*/
```

```
ndev->stats.rx_packets++; ndev->stats.rx_bytes += len;
```

```
/* Submit socket buffer to the network layer */
```

```
netif_rx_ni(skb);
```

```
}
```

```
}
```

```
/* Move the RX read pointer to the start of the next * received  
packet.
```

```
*/
```

```
priv->next_pk_ptr = my_netdev_update_reg_next_pkt(); }
```

Of course the only reason we call the RX handler from within a deferred work is because we sit on an SPI bus. All of the preceding operations could be performed from within the `hwirq` in case of an

MMIO device. Have a look at the NXP FEC driver, in `drivers/net/ethernet/freescale/fec.c` to see how this is achieved.

Packet transmission

When the kernel needs to send packets out of the interface, it calls the driver's `ndo_start_xmit` method, which should return `NETDEV_TX_OK` ON SUCCESS, OR `NETDEV_TX_BUSY` on failure, and in this case you can't do anything to the socket buffer since it is still owned by the network queuing layer when the error is returned. This means you cannot modify any SKB fields, or free the SKB, and so on. This function is protected from the concurrent call by a spinlock.

Packet transmission is done asynchronously in most cases. The `sk_buff` of the transmitted packet is filled by the upper layers. Its `data` field contains packets to be sent. Drivers should extract packet from `sk_buff->data` and write it into the device hardware FIFO, or put it into a temporary TX buffer (if the device needs a certain size of data before sending it) before writing it into the device hardware FIFO. Data is really only sent once the FIFO reaches a threshold value (usually defined by the driver, or provided in a device datasheet) or when the driver intentionally starts the transmission, by setting a bit (a kind of trigger) in a special register of the device. That being said, the driver needs to inform the kernel not to start any transmissions until the hardware is ready to accept new data. This notification is done by means of the `netif_stop_queue()` function.

```
| void netif_stop_queue(struct net_device *dev)
```

After sending the packet, the network interface card will raise an interrupt. The interrupt handler should check why the interrupt has occurred. In case of transmission interrupt, it should update its statistics (`net_device->stats.tx_errors` and `net_device->stats.tx_packets`), and notify the kernel that the device is free for sending new packets. This notification is done by means of `netif_wake_queue()`:

```
| void netif_wake_queue(struct net_device *dev)
```

To summarize, packet transmission is split into two parts:

- `ndo_start_xmit` operation, which notifies the kernel that the device is busy, set up everything, and starts the transfer.
- The TX interrupt handler, which updates TX statistics and notifies the kernel that the device is available again.

The `ndo_start_xmit` function must roughly contain the following steps:

1. Call `netif_stop_queue()` on the network device in order to inform the kernel that the device will be busy in data transmission.
2. Write `skb->data` content into the device FIFO.
3. Trigger the transmission (instruct the device to start transmission).



Operations (2) and (3) may lead to sleep for devices sitting on slow buses (SPI for example) and may need to be deferred to the work structure. This is the case for our sample.

Once the packet is transferred, the TX interrupt handler should perform the following steps:

4. Depending on the device being memory mapped or sitting on a bus whose access functions may sleep, the following operations should be performed directly in the hwirq handler or scheduled in a work (or threaded IRQ):
 1. Check if the interrupt is a transmission interrupt.
 2. Read the transmission descriptor status register and see what the status of the packet is.
 3. Increment error statistics if there are any problems in the transmission.
 4. Increment statistics of successful transmitted packets.
5. Start the transmission queue allowing the kernel to call the driver's `ndo_start_xmit` method again, by means of the `netif_wake_queue()` function.

Let us summarize in a short sample code:

```
/* Somewhere in the code */
INIT_WORK(&priv->tx_work, my_netdev_hw_tx);

static netdev_tx_t my_netdev_start_xmit(struct sk_buff *skb,
                                         struct net_device *dev)
{
    struct priv_net_struct *priv = netdev_priv(dev);

    /* Notify the kernel our device will be busy */
    netif_stop_queue(dev);
}
```

```

    /* Remember the skb for deferred processing */
    priv->tx_skb = skb;

    /* This work will copy data from sk_buffer->data to
     * the hardware's FIFO and start transmission
     */
    schedule_work(&priv->tx_work);

    /* Everything is OK */
    return NETDEV_TX_OK;
}
The work is described below:
/*
 * Hardware transmit function.
 * Fill the buffer memory and send the contents of the
 * transmit buffer onto the network
 */
static void my_netdev_hw_tx(struct priv_net_struct *priv)
{
    /* Write packet to hardware device TX buffer memory */
    my_netdev_packet_write(priv, priv->tx_skb->len,
priv->tx_skb->data);

    /*
     * does this network device support write-verify?
     * Perform it
     */
    [...];

    /* set TX request flag,
     * so that the hardware can perform transmission.
     * This is device-specific
     */
    my_netdev_reg_bitset(priv, ECON1, ECON1_TXRTS);
}

```

TX interrupt management will be discussed in the next section.

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

#include <linux/kernel.h>

#include <linux/errno.h>

#include <linux/init.h>

#include <linux/netdevice.h>

#include <linux/etherdevice.h>

#include <linux/ethtool.h>

#include <linux/skbuff.h>

#include <linux/slab.h>

#include <linux/of.h> /* For DT*/

#include <linux/platform_device.h> /* For platform devices */


struct eth_struct {

    int bar;

    int foo;

    struct net_device *dummy_ndev; };


static int fake_eth_open(struct net_device *dev) {
```

```
    printk("fake_eth_open called\n"); /* We are now ready to accept
transmit requests from * the queueing layer of the networking.
```

```
    */
```

```
    netif_start_queue(dev); return 0;
```

```
}
```

```
static int fake_eth_release(struct net_device *dev) {
```

```
    pr_info("fake_eth_release called\n"); netif_stop_queue(dev); return
0;
```

```
}
```

```
static int fake_eth_xmit(struct sk_buff *skb, struct net_device *ndev)
{
```

```
    pr_info("dummy xmit called...\n"); ndev->stats.tx_bytes += skb-
>len; ndev->stats.tx_packets++;
```

```
    skb_tx_timestamp(skb); dev_kfree_skb(skb); return
NETDEV_TX_OK; }
```

```
static int fake_eth_init(struct net_device *dev) {
```

```
    pr_info("fake eth device initialized\n"); return 0;
```

```
};
```

```
static const struct net_device_ops my_netdev_ops = {  
  
    .ndo_init = fake_eth_init, .ndo_open = fake_eth_open, .ndo_stop =  
    fake_eth_release, .ndo_start_xmit = fake_eth_xmit,  
    .ndo_validate_addr = eth_validate_addr, .ndo_validate_addr =  
    eth_validate_addr, };
```

```
static const struct of_device_id fake_eth_dt_ids[] = {  
  
    { .compatible = "packt,fake-eth", }, { /* sentinel */ }  
  
};
```

```
static int fake_eth_probe(struct platform_device *pdev) {  
  
    int ret;  
  
    struct eth_struct *priv; struct net_device *dummy_ndev;  
  
    priv = devm_kzalloc(&pdev->dev, sizeof(*priv), GFP_KERNEL);  
    if (!priv)  
  
        return -ENOMEM;  
  
  
  
    dummy_ndev = alloc_etherdev(sizeof(struct eth_struct));
```

```
dummy_ndev->if_port = IF_PORT_10BASET; dummy_ndev->netdev_ops = &my_netdev_ops;
```

```
/* If needed, dev->ethtool_ops = &fake_ethtool_ops; */
```

```
ret = register_netdev(dummy_ndev); if(ret) {  
    pr_info("dummy net dev: Error %d initializing card ...", ret); return  
ret;  
}
```

```
priv->dummy_ndev = dummy_ndev; platform_set_drvdata(pdev,  
priv); return 0;  
}
```

```
static int fake_eth_remove(struct platform_device *pdev) {  
    struct eth_struct *priv; priv = platform_get_drvdata(pdev);  
    pr_info("Cleaning Up the Module\n"); unregister_netdev(priv->dummy_ndev); free_netdev(priv->dummy_ndev);  
    return 0;  
}
```

```
static struct platform_driver mypdrv = {
    .probe = fake_eth_probe, .remove = fake_eth_remove, .driver = {
        .name = "fake-eth", .of_match_table =
of_match_ptr(fake_eth_dt_ids), .owner = THIS_MODULE, },
};

module_platform_driver(mypdrv);
```

```
MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
MODULE_DESCRIPTION("Fake Ethernet driver");
```

```
<strong># dmesg</strong><br/><strong>[...]</strong><br/><strong>
[146698.060074] fake eth device initialized</strong><br/><strong>
[146698.087297] IPv6: ADDRCONF(NETDEV_UP): eth0: link is
not ready</strong>
```

```
<strong># ifconfig -a</strong><br/><strong>[...]</strong><br/>
<strong>eth0 Link encap:Ethernet HWaddr
00:00:00:00:00:00</strong><br/><strong>BROADCAST
MULTICAST MTU:1500 Metric:1</strong><br/><strong>RX
packets:0 errors:0 dropped:0 overruns:0 frame:0</strong><br/>
<strong>TX packets:0 errors:0 dropped:0 overruns:0
carrier:0</strong><br/><strong>collisions:0
txqueuelen:1000</strong><br/><strong>RX bytes:0 (0.0 B) TX
bytes:0 (0.0 B)</strong>
```

```
<strong># ifconfig eth0 192.168.1.45</strong><br/><strong>#
ifconfig</strong><br/><strong>[...]</strong><br/><strong>eth0 Link
```


encap:Ethernet HWaddr 00:00:00:00:00:00

inet addr:192.168.1.45 Bcast:192.168.1.255
Mask:255.255.255.0
BROADCAST
MULTICAST MTU:1500 Metric:1
RX
packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0
carrier:0
collisions:0
txqueuelen:1000
RX bytes:0 (0.0 B) TX
bytes:0 (0.0 B)

Status and control

Device control refers to a situation where the kernel needs to change properties of the interface on its own initiative, or in response to a user action. It can then use either operations exposed through the `struct net_device_ops` structure, as discussed, or use another control tool, **ethtool**, which requires the driver to introduce a new set of hooks that we will discuss in the next section. Conversely, status consists of reporting the state of the interface.

```

static irqreturn_t my_netdev_irq(int irq, void *dev_id) {

    struct priv_net_struct *priv = dev_id;

    /*
     * Can't do anything in interrupt context because we need to * block
     (spi_sync() is blocking) so fire of the interrupt * handling workqueue.

     * Remember, we access our netdev registers through SPI bus * via
     spi_sync() call.
    */

    schedule_work(&priv->irq_work);

    return IRQ_HANDLED;
}

static void my_netdev_irq_work_handler(struct work_struct *work) {

    struct priv_net_struct *priv =

        container_of(work, struct priv_net_struct, irq_work); struct
    net_device *ndev = priv->netdev; int intflags, loop;

    /* disable further interrupts */

    my_netdev_reg_bitclear(priv, EIE, EIE_INTIE);

    do {

```

```

loop = 0;

intflags = my_netdev_regb_read(priv, EIR); /* DMA interrupt
handler (not currently used) */

if ((intflags & EIR_DMAIF) != 0) {

loop++;

handle_dma_complete(); clear_dma_interrupt_flag(); }

/* LINK changed handler */

if ((intflags & EIR_LINKIF) != 0) {

loop++;

my_netdev_check_link_status(ndev); clear_link_interrupt_flag(); }

/* TX complete handler */

if ((intflags & EIR_TXIF) != 0) {

bool err = false;

loop++;

priv->tx_retry_count = 0; if (locked_regb_read(priv, ESTAT) &
ESTAT_TXABRT) clear_tx_interrupt_flag();

/* TX Error handler */

if ((intflags & EIR_TXERIF) != 0) {

loop++;

```

```

/*

    * Reset TX logic by setting/clearing appropriate * bit in the right
    register */

    [...]

/* Transmit Late collision check for retransmit */

if (my_netdev_cpllosion_bit_set()) /* Handle collision */

    [...]

}

/* RX Error handler */

if ((intflags & EIR_RXERIF) != 0) {

    loop++;

    /* Check free FIFO space to flag RX overrun */

    [...]

}

/* RX handler */

if (my_rx_interrupt(ndevice)) loop++;

} while (loop);

```

```
/* re-enable interrupts */
```

```
my_netdev_reg_bitset(priv, EIE, EIE_INTIE); }
```

Ethtool support

Ethtool is a small utility for examining and tuning the settings of Ethernet-based network interfaces. With ethtool, it is possible to control various parameters like:

- Speed
- Media type
- Duplex operation
- Get/set eeprom register content
- Hardware check summing
- Wake-on-LAN, and so on.

Drivers that need support from ethtool should include `<linux/ethtool.h>`. It relies on the `struct ethtool_ops` structure which is the core of this feature, and contains a set of methods for ethtool operations support. Most of these methods are relatively straightforward; see `include/linux/ethtool.h` for the details.

For ethtool support to be fully part of the driver, the driver should fill an `ethtool_ops` structure and assign it to the `.ethtool_ops` field of the `struct net_device` structure.

```
| my_netdev->ethtool_ops = &my_ethtool_ops;
```

The macro `SET_ETHTOOL_OPS` can be used for this purpose too. Do note that your ethtool methods can be called even when the interface is down.

For example, the following drivers implement ethtool support:

- `drivers/net/ethernet/microchip/enc28j60.c`
- `drivers/net/ethernet/freescale/fec.c`
- `drivers/net/usb/rtl8150.c`

```
static const struct net_device_ops my_netdev_ops = {  
  
    .ndo_open = my_netdev_open, .ndo_stop = my_netdev_close,  
    .ndo_start_xmit = my_netdev_start_xmit, .ndo_set_rx_mode =  
    my_netdev_set_multicast_list, .ndo_set_mac_address =  
    my_netdev_set_mac_address, .ndo_tx_timeout =  
    my_netdev_tx_timeout, .ndo_change_mtu = eth_change_mtu,  
    .ndo_validate_addr = eth_validate_addr, };
```

The preceding are the operations that most drivers implement.


```
static int my_netdev_probe(struct spi_device *spi) {  
  
    struct net_device *dev;  
  
    struct priv_net_struct *priv; int ret = 0;  
  
    /* Allocate network interface */  
  
    dev = alloc_etherdev(sizeof(struct priv_net_struct)); if (!dev)  
    [...] /* handle -ENOMEM error */  
  
    /* Private data */  
  
    priv = netdev_priv(dev);  
  
    /* set private data and bus-specific parameter */  
  
    [...]   
  
    /* Initialize some works */  
  
    INIT_WORK(&priv->tx_work, data_tx_work_handler); [...]   
  
    /* Devicerealy init, only few things */
```

```

if (!my_netdev_chipset_init(dev)) [...] /* handle -EIO error */

/* Generate and assign random MAC address to the device */

eth_hw_addr_random(dev);

my_netdev_set_hw_macaddr(dev);

/* Board setup must set the relevant edge trigger type; * level
triggers won't currently work.

*/

ret = request_irq(spi->irq, my_netdev_irq, 0, DRV_NAME, priv); if
(ret < 0)

[...]; /* Handle irq request failure */

/* Fill some netdev mandatory or useful properties */

dev->if_port = IF_PORT_10BASET; dev->irq = spi->irq; dev-
>netdev_ops = &my_netdev_ops; dev->ethtool_ops =
&my_ethtool_ops;

/* Put device into sleep mode */

My_netdev_lowpower(priv, true);

/* Register our device with the kernel */

if (register_netdev(dev))

```

```
[...]; /* Handle registration failure error */  
  
dev_info(&dev->dev, DRV_NAME " driver registered\n");  
  
return 0;  
  
}
```

This whole chapter is heavily inspired by the enc28j60 from Microchip. You may have a look into its code in `drivers/net/ethernet/microchip/enc28j60.c`.

The `register_netdev()` function takes a completed `struct net_device` object and adds it to the kernel interfaces; 0 is returned on success and a negative error code is returned on failure. The `struct net_device` object should be stored in your bus device structure so that it can be accessed later. That being said, if your net device is part of a global private structure, it is that structure that you should register.

Do note that the duplicate device name may lead to registration failure.

```
static int my_netdev_remove(struct spi_device *spi) {  
  
    struct priv_net_struct *priv = spi_get_drvdata(spi);  
  
    unregister_netdev(priv->netdev); free_irq(spi->irq, priv);  
    free_netdev(priv->netdev);  
  
    return 0; }
```

The `unregister_netdev()` function removes the interface from the system, and the kernel can no longer call its methods; `free_netdev()` frees the memory used by the `struct net_device` structure itself along with the memory allocated for private data, as well as any internally allocated memory related to the network device. Do note that you should never free `netdev->priv` by yourself.

Summary

This chapter has explained everything needed to write an NIC device driver. Even if the chapter relies on a network interface sitting on an SPI bus, the principle is the same for USB or PCI network interfaces. One can also use the dummy driver provided for testing purposes. After this chapter, it is obvious NIC drivers will no longer be mystery to you.