

Linux Driver Development with Raspberry Pi[®]

Practical Labs

Copyright © 2021, Alberto Liberal. All rights reserved.

No part of this publication may be reproduced, photocopied, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher.

Linux is a registered trademark of Linus Torvalds. Other trademarks within this book are the property of their respective owners.

The kernel modules examples provided with this book are released under the GPL license. This license gives you the right to use, study, modify and share the software freely. However, when the software is redistributed, modified or unmodified, the GPL requires that you redistribute the software under the same license, with the source code.

Published by:

Alberto Liberal
aliberal@arroweurope.com

Graphic Design: Signo Comunicación Consultores SLU

Imprint: Independently published

About the Author

Alberto Liberal is a Field Application Engineer at Arrow Electronics with over 15 years of experience in embedded systems. For the last few years at Arrow, he has been supporting high-end processors and FPGAs. Alberto is a Linux fan and has presented numerous technical seminars and practical workshops about embedded Linux and Linux device drivers during the past few years. Alberto's other fields of expertise include multimedia SoCs and real-time operating systems (RTOS). He currently lives in Madrid, Spain, and his great passion is taking long walks with his daughter through the center of Madrid. He also enjoys reading about cinema and watching sci-fi movies.

Table of Contents

Preface.....	13
Chapter 1: Building the System	19
Bootloader	20
Linux kernel	22
System call interface and C runtime library	25
System shared libraries	26
Root filesystem	27
Linux boot process.....	28
Building an embedded Linux system for the Raspberry Pi.....	29
Raspberry Pi OS.....	30
Connect and set up hardware	32
Setting up an ethernet communication	36
Copying files to your Raspberry Pi	36
Building the Linux kernel	37
Chapter 2: The Linux Device and Driver Model.....	41
Bus core drivers.....	42
Bus controller drivers.....	44
Device drivers	44
Devices	45
The sysfs filesystem	47
The kobject infrastructure	49
Introduction to the Device Tree.....	53
Chapter 3: The Simplest Drivers	61
Licensing.....	62
LAB 3.1: "helloworld" module	62
Listing 3-1: helloworld_rpi3.c	63
Listing 3-2: Makefile	63
helloworld_rpi3.ko demonstration	64
LAB 3.2: "helloworld with parameters" module.....	65
Listing 3-3: helloworld_rpi3_with_parameters.c.....	66
Listing 3-4: Makefile	66

helloworld_rpi3_with_parameters.ko demonstration.....	67
Chapter 4: Character Drivers.....	69
LAB 4.1: "helloworld character" module	71
Registration and unregistration of character devices	72
Listing 4-1: helloworld_rpi3_char_driver.c.....	76
Listing 4-2: Makefile	78
Listing 4-3: ioctl_test.c	78
helloworld_rpi3_char_driver.ko demonstration	79
Creating device files with devtmpfs	79
LAB 4.2: "class character" module	80
Listing 4-4: helloworld_rpi3_class_driver.c	82
helloworld_rpi3_class_driver.ko demonstration	84
Miscellaneous character driver	85
Registering a minor number	85
LAB 4.3: "miscellaneous character" module	86
Listing 4-5: misc_rpi3_driver.c	87
misc_rpi3_driver.ko demonstration	88
Chapter 5: Platform Drivers.....	89
LAB 5.1: "platform device" module	91
Listing 5-1: hellokeys_rpi3.c.....	93
hellokeys_rpi3.ko demonstration	95
Documentation to interact with the hardware	96
Hardware naming convention	96
Pin control subsystem.....	97
GPIO controller driver	106
GPIO descriptor consumer interface	110
Obtaining and disposing GPIOs	110
Using GPIOs.....	110
GPIOs mapped to IRQs.....	111
GPIOs in the Device Tree	112
Exchanging data between kernel and user spaces	112
MMIO (Memory-Mapped I/O) device access	113
LAB 5.2: "RGB LED platform device" module	115

LAB 5.2 hardware description	116
LAB 5.2 Device Tree description	117
LAB 5.2 code description of the "RGB LED platform device" module	121
Listing 5-2: ledRGB_rpi3_platform.c	126
ledRGB_rpi3_platform.ko demonstration	131
Platform driver resources	132
Linux LED class	133
LAB 5.3: "RGB LED class" module	136
LAB 5.3 Device Tree description	136
LAB 5.3 code description of the "RGB LED class" module	137
Listing 5-3: ledRGB_rpi3_class_platform.c	141
ledRGB_rpi3_class_platform.ko demonstration	145
Platform device drivers in user space	146
User defined I/O: UIO	148
How UIO works	150
Kernel UIO API	151
LAB 5.4: "LED UIO platform" module	153
LAB 5.4 Device Tree description	153
LAB 5.4 code description of the "LED UIO platform" module	153
Listing 5-4: led_rpi3_UIO_platform.c	156
Listing 5-5: UIO_app.c	157
led_rpi3_UIO_platform.ko with UIO_app demonstration	160
Chapter 6: I2C Client Drivers	161
The Linux I2C subsystem	163
Writing I2C client drivers	165
I2C client driver registration	165
Declaration of I2C devices in the Device Tree	167
LAB 6.1: "I2C I/O expander device" module	168
LAB 6.1 hardware description	168
LAB 6.1 Device Tree description	168
LAB 6.1 code description of the "I2C I/O expander device" module	169
Listing 6-1: io_rpi3_expander.c	173
io_rpi3_expander.ko demonstration	176
LAB 6.2: "I2C multidisplay LED" module	177
LAB 6.2 hardware description	178
LAB 6.2 Device Tree description	178

Unified device properties interface for ACPI and Device Tree.....	180
LAB 6.2 code description of the "I2C multidisplay LED" module.....	181
Listing 6-2: ltc3206_rpi3_led_class.c.....	186
ltc3206_rpi3_led_class.ko demonstration.....	192
Chapter 7: Handling Interrupts in Device Drivers	195
Linux kernel IRQ domain for GPIO controllers	198
Device Tree interrupt handling.....	204
Requesting interrupts in Linux device drivers	206
LAB 7.1: "button interrupt device" module	207
LAB 7.1 hardware description	207
LAB 7.1 Device Tree description	208
LAB 7.1 code description of the "button interrupt device" module.....	209
Listing 7-1: int_rpi3_key.c	211
int_rpi3_key.ko demonstration	213
Deferred work	214
Softirqs	215
Tasklets.....	217
Timers.....	217
Threaded interrupts	221
Workqueues	223
Locking in the kernel.....	226
Locks and uniprocessor kernels	227
Sharing spinlocks between interrupt and process context	227
Locking in user context.....	227
Sleeping in the kernel	228
LAB 7.2: "sleeping device" module.....	230
LAB 7.2 Device Tree description	230
LAB 7.2 code description of the "sleeping device" module	232
Listing 7-2: int_rpi3_key_wait.c.....	235
int_rpi3_key_wait.ko demonstration.....	238
Kernel threads	239
LAB 7.3: "keyled class" module	240
LAB 7.3 hardware description	241
LAB 7.3 Device Tree description	241
LAB 7.3 code description of the "keyled class" module.....	242
Listing 7-3: keyled_rpi3_class.c.....	249

keyled_rpi3_class.ko demonstration	259
LAB 7.4: "GPIO expander device" module	262
LAB 7.4 hardware description	263
LAB 7.4 Device Tree description	264
LAB 7.4 GPIO controller driver description	265
Listing 7-4: CY8C9520A_rpi3.c	271
LAB 7.4 GPIO child driver description	283
Listing 7-5: int_rpi3_gpio.c	285
LAB 7.4 GPIO based IRQ application	286
Listing 7-6: gpio_int.c	287
LAB 7.4 driver demonstration	289
LAB 7.5: "GPIO-PWM-PINCTRL expander device" module	293
LAB 7.5 GPIO irqchip description	293
LAB 7.5 pin controller driver description	295
LAB 7.5 PWM controller driver description	299
Listing 7-7: CY8C9520A_pwm_pinctrl.c	302
LAB 7.5 driver demonstration	323
LAB 7.6: CY8C9520A Device Tree overlay	325
LAB 7.6 driver demonstration	332
Chapter 8: Allocating Kernel Memory	333
Linux address types	333
User process virtual to physical memory mapping	335
Kernel virtual to physical memory mapping	336
Kernel memory allocators	337
Page allocator	338
Page allocator API	338
SLAB allocator	339
SLAB allocator API	342
Kmalloc allocator	343
LAB 8.1: "linked list memory allocation" module	344
Listing 8-1: linkedlist_rpi3_platform.c	346
linkedlist_rpi3_platform.ko demonstration	351
Chapter 9: DMA in Device Drivers	353
Cache coherency	353
Linux DMA Engine API	355
Types of DMA mappings	357

LAB 9.1: "streaming DMA" module	361
Listing 9-1: sdma_rpi3_m2m.c	366
sdma_rpi3_m2m.ko demonstration	370
Chapter 10: Input Subsystem	371
Input subsystem drivers	372
LAB 10.1: "input subsystem accelerometer" module	374
Device Tree	376
Input framework as an I2C interaction	376
Input framework as an input device	378
Listing 10-1: i2c_rpi3_accel.c	381
i2c_rpi3_accel.ko demonstration	383
LAB 10.2: "Nunchuk input subsystem" module	384
LAB 10.2 hardware description	385
LAB 10.2 Device Tree description	386
LAB 10.2 Nunchuk controller driver description	387
Listing 10-2: nunchuk.c	393
nunchuk.ko demonstration	398
LAB 10.3: Nunchuk applications	400
joystick_led.py application	403
Listing 10-3: joystick_led.py	403
joystick_led.py demonstration	404
joystick_pwm.py application	405
Listing 10-4: joystick_pwm.py	405
joystick_pwm.py demonstration	406
joystick_pygame.py application	407
Listing 10-5: joystick_pygame.py	407
joystick_pygame.py demonstration	409
Using SPI with Linux	411
The Linux SPI subsystem	413
Writing SPI client drivers	416
SPI client driver registration	416
Declaration of SPI devices in the Device Tree	417
LAB 10.4: "SPI accel input device" module	419
LAB 10.4 hardware description	419
LAB 10.4 Device Tree description	420

LAB 10.4 code description of the "SPI accel input device" module	420
Listing 10-6: adxl345_rpi3.c	430
adxl345_rpi3.ko demonstration	442
Chapter 11: Industrial I/O Subsystem	445
IIO device sysfs interface	447
IIO device channels	447
The iio_info structure	449
Buffers	450
IIO buffer sysfs interface	450
IIO buffer setup	450
Triggers	452
Triggered buffers	452
Industrial I/O events	454
Delivering IIO events to user space	456
IIO utils	458
LAB 11.1: "IIO Mixed-Signal I/O Device" module	458
LAB 11.1 hardware description	461
LAB 11.1 Device Tree description	463
LAB 11.1 driver description	468
Listing 11-1: max11300-base.h	483
Listing 11-2: maxim,max11300.h	485
Listing 11-3: max11300.c	486
Listing 11-4: max11300-base.c	489
LAB 11.1 driver demonstration	507
GPIO control through a character device	511
Listing 11-5: gpio_device_app.c	512
GPIO control through the gpiolibd library	513
Listing 11-6: libgpiod_max11300_app.c	514
LAB 11.2: "Nunchuk provider and consumer" modules	515
Nunchuk provider module	515
Listing 11-7: nunchuk_accel.c	519
Nunchuk consumer module	522
Listing 11-8: nunchuk_consumer.c	523
LAB 11.2 Device Tree description	527
LAB 11.2 driver demonstration	529

Chapter 12: Using the Regmap API in Device Drivers	531
LAB 12.1: "SPI regmap IIO device" module	537
Listing 12-1: adxl345_rpi3_iio.c	547
adxl345_rpi3_iio.ko demonstration	558
Chapter 13: USB Device Drivers	563
USB 2.0 bus topology	563
USB bus enumeration and device layout.....	564
USB data transfers	567
USB device classes	568
Human interface device class.....	568
USB descriptors.....	569
USB device descriptors	569
USB configuration descriptor.....	571
USB interface descriptor.....	572
USB endpoint descriptor.....	573
USB string descriptors.....	573
USB HID descriptor	574
The Linux USB subsystem	576
Writing Linux USB device drivers	577
USB device driver registration	578
Linux host-side data types	579
USB request block (URB).....	581
LAB 13.1: USB HID Device Application	584
STEP 1: Create a new project.....	585
STEP 2: Configure Harmony	586
STEP 3: Modify the generated code.....	589
STEP 4: Declare the USB State Machine states.....	589
STEP 5: Add new members to APP_DATA type.....	590
STEP 6: Declare the reception and transmission buffers.....	591
STEP 7: Initialize the new members.....	591
STEP 8: Handle the detach	592
STEP 9: Handle the HID events	592
STEP 10: Create the USB State Machine	594
STEP 11: Schedule a new report receive request.....	596
STEP 12: Receive, prepare and send reports.....	597
STEP 13: Program the application	598

LAB 13.2: "USB LED" module	598
LAB 13.2 code description of the "USB LED" module	599
Listing 13-1: usb_led.c	602
usb_led.ko demonstration	604
LAB 13.3: "USB LED and Switch" module	605
LAB 13.3 code description of the "USB LED and Switch" module	606
Listing 13-2: usb_urb_int_led.c	610
usb_urb_int_led.ko demonstration	615
LAB 13.4: "I2C to USB Multidisplay LED" module	616
LAB 13.4 code description of the "I2C to USB Multidisplay LED" module	622
Listing 13-3: usb_ltc3206.c	627
usb_ltc3206.ko demonstration	633
References	635
Index	637

Preface

Embedded systems have become an integral part of our daily life. They are deployed in mobile devices, networking infrastructure, home and consumer devices, digital signage, medical imaging, automotive infotainment and many other industrial applications. The use of embedded systems is growing exponentially. Many of these embedded systems are powered by an inexpensive yet powerful system-on-chip (SoC) that is running a Linux operating system. The BCM2837 from Broadcom is one of these SoCs, running quad ARM Cortex A53 cores at 1.2GHz. This is the SoC used in the popular Raspberry Pi 3 boards.

This book follows the learning by doing approach, so you will be playing with your Raspberry Pi since the first chapter. Besides the Raspberry Pi board, you will use several low-cost boards to develop the hands-on examples. In the labs, it is described what each step means in detail so that you can use your own hardware components adapting the content of the book to your needs.

You will learn how to develop Linux drivers for the Raspberry Pi boards. You will start with the simplest ones that do not interact with any external hardware, then you will develop Linux drivers that manage different kind of devices: Accelerometer, DAC, ADC, RGB LED, Buttons, Joystick controller, Multi-Display LED controller and I/O expanders controlled via I2C and SPI buses. You will also develop DMA drivers, USB device drivers, drivers that manage interrupts and drivers that write and read on the internal registers of the SoC to control its GPIOs. To ease the development of some of these drivers, you will use different types of Linux kernel subsystems: Miscellaneous, LED, UIO, USB, Input and Industrial I/O. More than 30 kernel modules have been written (besides several user applications), which can be downloaded from the book's GitHub repository.

This book uses the Long Term Support (LTS) Linux kernel 5.4, which was released on November 2019 and will be maintained until December 2025.

This book is a learning tool to start developing drivers without any previous knowledge about this field, so the intention during its writing has been to develop drivers without a high level of complexity that both serve to reinforce the main driver development concepts and can be a starting point to help you to develop your own drivers. And, remember that the best way to develop a driver is not to write it from scratch. You can reuse free code from similar Linux kernel mainline drivers. All the drivers written throughout this book are GPL licensed, so you can modify and redistribute them under the same license.

How this book is structured

Chapter 1, Building the System, starts by describing the main parts of an embedded Linux system. Next, it describes in detail how to build an embedded Linux system for the Raspberry Pi. You will install on a Micro SD a Raspberry Pi OS image based on kernel 5.4.y, then you will download the Raspberry Pi kernel sources from the rpi-5.4.y branch, configure the kernel with the settings needed to develop the drivers, build the kernel, and finally, you will install the new kernel on the Raspberry Pi Linux image. The generated Linux image will be used for the testing of drivers and applications throughout the book.

Chapter 2, The Linux Device and Drivel Model, explains the relationship between "Bus" drivers, "Bus controller" drivers and "Device" drivers. It also provides an introduction to the Device Tree and the sysfs filesystem.

Chapter 3, The Simplest Drivers, covers several simple drivers that are not yet interacting with user applications through "system calls". This chapter will let you check that your driver development ecosystem is working properly.

Chapter 4, Character Drivers, describes the architecture of the character drivers. It explains how driver's operations are called from user space by using system calls and how to exchange data between kernel and user spaces. It also explains how to identify and create each Linux device. Several drivers will be written that exchange information with user space using different methods for the creation of the device nodes. The first driver in this chapter will use the traditional static device creation method by using the "mknod" command, the second one will show how create device files with "devtmpfs", and the last one will create the device files by using the "miscellaneous framework". This chapter also explains how to create a device class and a device driver entry under the sysfs.

Chapter 5, Platform Drivers, describes what a platform driver is, how a platform device is statically described in the Device Tree and the process of associating the device with the device driver, called "binding". In this chapter, you will develop your first driver interacting with the hardware. Before developing the driver, a detailed explanation will be provided on the way the pads of a Raspberry Pi SoC can be multiplexed to the different functions and how to select the required pin muxing option within the Device Tree. This chapter also describes the Pinctrl Subsystem and the GPIO Descriptor Consumer Interface. You will develop drivers that control GPIOs writing and reading registers on the SoC. You will also learn to write drivers that control LEDs by using the Linux LED subsystem. Finally, this chapter explains how to develop a user space driver by using the UIO subsystem.

Chapter 6, I2C Client Drivers, describes the Linux I2C subsystem, which is based on the Linux device model. You will learn to declare I2C devices in the Device Tree, and you will develop

several I2C client drivers throughout this chapter. You will also see how to add "sysfs" support to control the hardware from user space via sysfs entries.

Chapter 7, Handling Interrupts in Device Drivers, introduces interrupt hardware and software operation on embedded processors running Linux. You will learn about the different structures used to create Linux drivers for interrupt controllers and the different types of GPIO irqchips. You will develop several drivers that manage interrupts coming from the GPIOs of the Raspberry Pi SoC. In one of these drivers, you will see how a user application is put to sleep by using a "wait queue" and woken up later via an interrupt. You will also develop a GPIO controller driver with interrupt capabilities for an off-chip GPIO expander and a user application that requests interrupts from the GPIO expander by using GPIOlib APIs. In this chapter, you will see how to introduce new hardware support by using the Raspberry Pi Device Tree overlay mechanism.

Chapter 8, Allocating Kernel Memory, explains the different types of addresses used in Linux. This chapter also describes the different kernel memory allocators.

Chapter 9, DMA in Device Drivers, describes the "Linux DMA Engine Subsystem" and the different types of DMA mappings. You will develop a DMA driver for the Raspberry Pi that manages memory to memory transactions without CPU intervention.

Chapter 10, Input Subsystem, introduces the use of frameworks to provide a coherent user space interface for every type of device, regardless of the drivers. The chapter explains in the relationship between the physical and logical parts of a driver that uses a kernel framework. It focuses on the Input subsystem, which takes care of the input events coming from the human user. In this chapter, you will develop a driver for the Wii Nunchuk controller and several Python applications that interact with the events generated by the Nunchuk driver. One of these applications will use the Pygame Python modules, drawing the Nunchuk Joystick values on a screen connected to the Raspberry Pi board. This chapter also describes the Linux SPI subsystem, which is based on the Linux device model. You will develop an SPI client driver for an accelerometer device using the Input subsystem.

Chapter 11, Industrial I/O Subsystem, describes the IIO subsystem, which provides support for ADCs, DACs, gyroscopes, accelerometers, magnetometers, proximity sensors, etc. The set up of IIO triggered buffers and Industrial I/O events will be explained in detail. You will create an IIO driver to control the Maxim MAX11300 device, which integrates a PIXI™, 12-bit, multichannel, analog-to-digital converter (ADC) and a 12-bit, multichannel, buffered digital-to-analog converter (DAC) in a single integrated circuit (IC). In this driver, you will create a GPIO controller, which configures and controls the MAX11300 ports. You will also develop one application that controls a GPIO of the MAX11300 device through a character device and another application that controls the same GPIO using the functions of the gpiolibd library. In this chapter, you will also develop two drivers using a provider-consumer implementation: the first one is an IIO provider driver

which will read the 3-axis data from the Nunchuk's accelerometer, and the second one is an Input subsystem consumer driver which will read the IIO channel values from the IIO provider driver and report them to the Input subsystem.

Chapter 12, Using the Regmap API in Device Drivers, provides an overview of the regmap API and explains how it takes care of calling the relevant calls of the SPI or I2C subsystem. You will transform the SPI Input subsystem driver of the Chapter 10, which uses specific SPI core APIs, into an SPI IIO subsystem driver which uses the regmap API, keeping the same functionality between both drivers. Finally, you will dive into the "IIO tools" applications to test the driver.

Chapter 13, USB Device Drivers, describes the Linux USB subsystem, which is based on the Linux device model. You will learn how to create custom USB HID devices based on the Microchip PIC32MX microcontroller that will send/receive data to/from a Raspberry Pi Linux USB device driver. You will learn about main Linux USB data structures and functions and develop several Linux USB device drivers throughout this chapter.

Download the kernel modules

You can download the Linux kernel modules for this book from the GitHub repository at https://github.com/ALIBERA/linux_raspberrypi_book.

What you need to develop the drivers

The Linux drivers and applications developed in the labs have been ported to three different Raspberry Pi boards: Raspberry Pi 3 Model B, Raspberry Pi 3 Model B+ and Raspberry Pi 4 Model B. The specifications of these boards can be found at:

<https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>

<https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>

<https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>

In this book, the demonstration sections and Device Tree settings used during the development of the labs are referred to the Raspberry Pi 3 Model B board. You can download the Linux drivers, applications and Device Tree files for the three Raspberry Pi boards from the GitHub of the book. Besides the Raspberry Pi board, to get the most out of the book (by working through the labs), you will need the following hardware:

1. MikroElektronika Color click accessory board (used in LAB 5.2, LAB 7.3, LAB 10.3 and LAB 11.1):
<https://www.mikroe.com/color-click>

2. PCF8574 IO Expansion Board (used in LAB 6.1):
<https://www.waveshare.com/pcf8574-io-expansion-board.htm>
3. Analog Devices LTC3206 I2C Multidisplay board DC749A (used in LAB 6.2 and LAB 13.4):
<https://www.analog.com/en/products/ltc3206.html#product-evaluationkit>
4. MikroElektronika Button R click board (used in LAB 7.1, LAB 7.2 and LAB 7.3):
<https://www.mikroe.com/button-r-click>
5. MikroElektronika EXPAND 6 click board (used in LAB 7.4, LAB 7.5 and LAB 7.6):
<https://www.mikroe.com/expand-6-click>
6. MikroElektronika Accel click board (used in LAB 10.1, LAB 10.4 and LAB 12.1):
<https://www.mikroe.com/accel-click>
7. Olimex MOD-Wii-UEXT-NUNCHUCK (used in LAB 10.2, LAB 10.3 and LAB 11.2):
<https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/open-source-hardware>
8. MikroElektronika PIXI click board (used in LAB 11.1):
<https://www.mikroe.com/pixi-click>
9. Microchip Curiosity PIC32MX470 Development Board (used in LAB 13.1, LAB 13.2, LAB 13.3 and LAB 13.4):
<https://www.microchip.com/DevelopmentTools/ProductDetails/dm320103#additional-summary>

Questions

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

1

Building the System

The Linux kernel is one of the largest and most successful open source projects that has ever come about. The huge rate of change and the number of individual contributors shows that it has a vibrant and active community, constantly stimulating evolution of the kernel. This rate of change continues to increase, as does the number of developers and companies involved in the process. The development process has proved that it is able to scale up to higher speeds without trouble. The Linux kernel together with GNU software and many other open-source components provides a completely free operating system, GNU/Linux. Embedded Linux is the usage of the Linux kernel and various open-source components in embedded systems.

Embedded Linux is used in embedded systems such as consumer electronics (e.g., set-top boxes, smart TVs, PVRs (personal video recorders), IVI (in-vehicle infotainment), networking equipment (such as routers, switches, WAPs (wireless access points) or wireless routers), machine control, industrial automation, navigation equipment, spacecraft flight software, and medical instruments in general).

There are many advantages of using Linux in embedded systems. The following list shows some of these benefits:

- The main advantage of Linux is the ability to reuse components. Linux provides scalability due to its modularity and configurability.
- Open source. No royalties or licensing fees.
- Ported to a broad range of hardware architectures, platforms and devices.
- Broad support of applications and communication protocols (e.g., TCP/IP stack, USB stack, graphical toolkit libraries).
- Large support coming from an active community of developers.

These are the main components of an embedded Linux system: **Bootloader**, **Kernel**, **System call interface**, **C-Runtime library**, **System shared libraries** and **Root filesystem**. Each of these components will be described in more detail in the next sections.

Bootloader

Linux cannot be started on an embedded device without a small amount of machine specific code to initialize the system. Linux requires the bootloader code to do very little, although several bootloaders do provide extensive additional functionality. The minimal requirements are:

- Configuration of the memory system.
- Loading of the kernel image and the Device Tree at the correct addresses.
- Optional loading of an initial RAM disk at the correct memory address.
- Setting of the kernel command-line and other parameters (e.g., Device Tree, machine type).

It is also usually expected that the bootloader initializes a serial console for the kernel in addition to these basic tasks.

There are different bootloader options that come in all shapes and sizes. U-Boot is the standard bootloader for ARM Linux. U-Boot source code is located at <https://source.denx.de/u-boot/u-boot>.

These are some of the main U-Boot features:

1. **Small:** U-Boot is a bootloader, and its primary purpose in the system is to load an operating system. That means that U-Boot is necessary to perform a certain task, but it is not worth spending significant resources on. Typically, U-Boot is stored in the relatively small NOR flash memory, which is expensive compared to the much larger NAND devices normally used to store the operating system and the application. A usable and useful configuration of U-Boot, including a basic interactive command interpreter, support for download over Ethernet and the capability to program the flash should fit in no more than 128 KB.
2. **Fast:** The end user is not interested in running U-Boot. In most embedded systems, they are not even aware that U-Boot exists. The user wants to run some application code, and they want to do that as soon as possible after switching on the device. Initialize devices only when they are needed within U-Boot, i.e., don't initialize the Ethernet interface(s) unless U-Boot performs a download over Ethernet; don't initialize any IDE or USB devices unless U-Boot actually tries to load files from these, etc.
3. **Portable:** U-Boot is a bootloader, but it is also a tool used for board bring-up, for production testing and for other activities that are very closely related to hardware development. So far, it has been ported to several hundreds of different boards on about 30 different processor families.
4. **Configurable:** U-Boot is a powerful tool with many, many extremely useful features. The maintainer or user of each board will have to decide which features are important

and what shall be included with their specific board configuration to meet the current requirements and restrictions.

5. **Debuggable:** U-Boot is not only a tool in itself, it is often also used for hardware bring-up, so debugging U-Boot often means that you don't know if you are tracking down a problem in the U-Boot software or in the hardware you are running on. Code that is clean and easy to understand and debug is all the more important to everyone. One important feature of U-Boot is to enable output to the (usually serial) console as soon as possible in the boot process, even if this causes tradeoffs in other areas like memory footprint. All initialization steps shall print some "begin doing this" message before they actually start and some "done" message when they complete. The purpose of this is that you can always see which initialization step was running if any problems occur. This is important not only during software development, but also for the service people dealing with broken hardware in the field. U-Boot should be debuggable with simple JTAG or BDM equipment. It should use a simple, single-threaded execution model.

Linux kernel

Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance.

It has all the features you would expect in a modern, fully-fledged Unix implementation, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management and multistack networking including IPv4 and IPv6. Although originally developed for 32-bit x86-based PCs (386 or higher), today Linux also runs on a multitude of other processor architectures, in both 32-bit and 64-bit variants.

The Linux kernel is the lowest level of software running on a Linux system. It is charged with managing the hardware, running user programs and maintaining the overall security and integrity of the whole system. It is this kernel, which, after its initial release by Linus Torvalds in 1991, jump-started the development of Linux as a whole. The kernel is a relatively small part of the software on a full Linux system (many other large components come from the GNU project, the GNOME and KDE desktop projects, the X.org project and many other sources), but the kernel is the core which determines how well the system will work and is the piece which is truly unique to Linux.

The kernel, which forms the core of the Linux system, is the result of one of the largest cooperative software projects ever attempted. Regular two-to-three month releases deliver stable updates to Linux users, each with significant new features, added device support and improved performance. The rate of change in the kernel is high and increasing, with over 10,000 patches going into each recent kernel release. Each of these releases contains the work of more than 1,600 developers representing over 200 corporations.

As kernels move from the **mainline** into the **stable** category, two things can happen:

1. They can reach **End of Life** after a few bugfix revisions, which means that kernel maintainers will release no more bugfixes for this kernel version, or
2. They can be put into **longterm** maintenance, which means that maintainers will provide bugfixes for this kernel revision for a much longer period of time.

If the kernel version you are using is marked **EOL**, you should consider upgrading to the next major version, as there will be no more bugfixes provided in the kernel version you are using.

Linux kernel is released under **GNU GPL version 2** and is therefore Free Software as defined by the Free Software Foundation. You may read the entire copy of the license in the COPYING file distributed with each release of the Linux kernel.

Some of the subsystems the kernel is comprised of are listed below:

- **/arch/<arch>**: Architecture specific code.
- **/arch/<arch>/<mach>**: Machine/board specific code.
- **/Documentation**: Kernel documentation. Do not miss it!
- **/ipc**: Inter process communication.
- **/mm**: Memory management.
- **/fs**: File systems.
- **/include**: Linux kernel headers.
- **/include/asm-<arch>**: Architecture and machine dependent headers.
- **/include/linux**: Linux kernel core headers.
- **/init**: Linux initialization (including main.c).
- **/block**: Linux kernel block layer code.
- **/net**: Networking functionality.
- **/lib**: Common kernel helper functions.
- **/kernel**: Common kernel structures.
- **/arch**: Architecture specific code.
- **/crypto**: Cryptography code.
- **/security**: Security components.
- **/drivers**: Built-in drivers (does not include loadable modules).
- **Makefile**: Top Linux makefile (sets arch and version).
- **/scripts**: Scripts for internal or external use.

The official home for the mainline Linux kernel source code is www.kernel.org. You can download the source code either directly from the kernel.org website as a compressed tar.xz file, or you can download it through git, the kernel's preferred source code control system.

There are several main categories into which kernel releases may fall:

1. **Prepatch**: Prepatch or "RC" kernels are mainline kernel pre-releases that are mostly aimed at other kernel developers and Linux enthusiasts. They must be compiled from source and usually contain new features that must be tested before they can be put into a stable release. Prepatch kernels are maintained and released by Linus Torvalds.
2. **Mainline**: The mainline tree is maintained by Linus Torvalds. It's the tree where all new features are introduced and where all the exciting new development happens. New mainline kernels are released every 2-3 months.
3. **Stable**: After each mainline kernel is released, it is considered "stable". Any bug fixes for a stable kernel are backported from the mainline tree and applied by a designated stable kernel maintainer. There are usually only a few bugfix kernel releases until next mainline

kernel becomes available -- unless it is designated a "longterm maintenance kernel". Stable kernel updates are released on as-needed basis, usually 2-3 a month.

4. **Longterm:** There are usually several "longterm maintenance" kernel releases provided for the purposes of backporting bugfixes for older kernel trees. Only important bugfixes are applied to such kernels, and they don't usually see very frequent releases, especially for older trees.

In the following image, extracted from www.kernel.org, you can see the latest stable kernel, kernels under development (mainline, next), stable and long term kernels.

Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

Latest Release

5.12.4



mainline:	5.13-rc1	2021-05-09	[tarball]	[patch]	[view diff]	[browse]
stable:	5.12.4	2021-05-14	[tarball]	[pgp] [patch] [inc. patch]	[view diff]	[browse] [changelog]
stable:	5.11.21	2021-05-14	[tarball]	[pgp] [patch] [inc. patch]	[view diff]	[browse] [changelog]
longterm:	5.10.37	2021-05-14	[tarball]	[pgp] [patch] [inc. patch]	[view diff]	[browse] [changelog]
longterm:	5.4.119	2021-05-14	[tarball]	[pgp] [patch] [inc. patch]	[view diff]	[browse] [changelog]
longterm:	4.19.190	2021-05-07	[tarball]	[pgp] [patch] [inc. patch]	[view diff]	[browse] [changelog]
longterm:	4.14.232	2021-04-28	[tarball]	[pgp] [patch] [inc. patch]	[view diff]	[browse] [changelog]
longterm:	4.9.268	2021-04-28	[tarball]	[pgp] [patch] [inc. patch]	[view diff]	[browse] [changelog]
longterm:	4.4.268	2021-04-28	[tarball]	[pgp] [patch] [inc. patch]	[view diff]	[browse] [changelog]
linux-next:	next-20210514	2021-05-14				[browse]

In addition to the official versions of the kernel, there are many third-parties (chip-vendors, sub-communities) that supply and maintain their own version of the kernel sources by forking from the official kernel tree. The intent is to separately develop support for a particular piece of hardware or subsystem and to integrate this support to the official kernel at a later point. This process is called mainlining and describes the task to integrate the new feature or hardware support to the upstream (official) kernel. These are called **Distribution kernels**.

It is easy to tell if you are running a Distribution kernel. Unless you downloaded, compiled and installed your own version of kernel from www.kernel.org, you are running a Distribution kernel. To find out the version of your kernel, run `uname -r` after booting the processor.

You will work with the **longterm kernel 5.4.y** releases to develop all the drivers throughout this book.

System call interface and C runtime library

The system call is the fundamental interface between an application and the Linux kernel. System calls are the only means by which a user space application can interact with the kernel. In other words, they are the bridge between user space and kernel space. The strict separation of user and kernel space ensures that user space programs cannot freely access kernel internal resources, thereby ensuring the security and stability of the system. The system calls elevate the privilege of the user process.

The system call interface is generally not invoked directly (even though it could be) but rather through wrapper functions in the C runtime library. Some of these wrapper functions are only very thin layers over the system call (just checking and setting the call parameters) while others add additional functionality. The following image shows some system calls and their descriptions:

System Call	Description
Insmod (system)	Load driver module
Open	Open device
Read	Read from device
Write	Write to device
Close	Close device
Rmmod (system)	Unregister driver module

The C runtime library (C-standard library) defines macros, type definitions and functions for string handling, mathematical functions, input/output processing, memory allocation and several other functions that rely on OS services. The runtime library provides applications with access to OS resources and functions by abstracting the OS System call interface.

Several C runtime libraries are available: glibc, uClibc, eglibc, dietlibc, newlib. The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific C library.

The GNU C library, **glibc**, is the default C library used for example in the Yocto project. The GNU C Library is primarily designed to be a portable and high performance C library. It follows all relevant standards including ISO C11 and POSIX.1-2008. It is also internationalized and has

one of the most complete internationalization interfaces known. You can find the glibc manual at <https://www.gnu.org/software/libc/manual/>.

System shared libraries

System shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. System shared libraries are typically linked with a user space application to provide its access to a specific system functionality. This system functionality can be either self-contained like compression or encryption algorithms or require access to underlying kernel resources or hardware. In the latter case, the library provides a simple API that abstracts the complexities of the kernel or direct driver access.

In other words, system shared libraries encapsulate system functionality and therefore are an essential building block when building applications that interact with the system.

Every shared library has a special name called the "soname". The soname has the prefix "lib", the name of the library, the phrase ".so", followed by a period and a version number that is incremented whenever the interface changes (as a special exception, the lowest-level C libraries don't start with "lib"). A fully-qualified soname includes as a prefix the directory it's in. In a working system, a fully-qualified soname is simply a symbolic link to the shared library's "real name".

Every shared library also has a "real name", which is the filename containing the actual library code. The real name adds a period to the soname, a minor number, another period and the release number. The last period and the release number are optional. The minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed. Note that these numbers might not be the same as the numbers used to describe the library in documentation, although that does make things easier.

In addition, there's the name that the compiler uses when requesting a library, (call it the "linker name"), which is simply the soname without any version number.

The following system shared libraries are required by the LSB (Linux Standard Base) specification and therefore must be available on all LSB compliant systems:

- **Libc:** Standard C library (C runtime library). Elementary language support and OS platform services. Direct access to the OS System-Call-Interface.
- **Libm:** Math Library. Common elementary mathematical functions and floating point environment routines defined by System V, ANSI C, POSIX, etc.

- **Libpthread:** POSIX thread library. Functionality now in libc, maintained to provide backwards compatibility.
- **Libdl:** Dynamic Linker Library. Functionality now in libc, maintained to provide backwards compatibility.
- **Libcrypt:** Cryptology Library. Encryption and decryption handling routines.
- **Libpam:** PAM (Pluggable Authentication Module) library. Routines for the PAM.
- **Libz:** Compression/decompression library. General purpose data compression and deflation functionality.
- **Libncurses:** CRT screen handling and optimization package. Overall screen, window and pad manipulation; output to windows and pads; reading terminal input; control over terminal and cursor input and output options; environment query routines; color manipulation; use of soft label keys.
- **Libutil:** System utilities library. Various system-dependent utility routines used in a wide variety of system daemons. The abstracted functions are mostly related to pseudo-terminals and login accounting.

Libraries are placed in the following standard root filesystem locations:

- **/lib:** Libraries required for startup.
- **/usr/lib:** Most system libraries.
- **/usr/local/lib:** Non-system libraries.

Note: A large part of the text of this section has been extracted from the Linux Documentation Project. You can find all the complete information at <https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html>.

Root filesystem

The root filesystem is where all the files contained in the file hierarchy (including device nodes) are stored. The root filesystem is mounted as `/`, containing all the libraries, applications and data.

The folder structure of the root filesystem is defined by FHS (Filesystem-Hierarchy-Standard). The FHS defines the names, locations and permissions for many file types and directories. It thereby ensures compatibility between different Linux distributions and allows applications to make assumptions about where to find specific system files and configurations.

An embedded Linux root filesystem, usually includes the following:

- **/bin:** Commands needed during bootup that might be used by normal users (probably after bootup).

- **/sbin:** Like /bin, but the commands are not intended for normal users, although they may use them if necessary and allowed; /sbin is not usually in the default path of normal users, but will be in root's default path.
- **/etc:** Configuration files specific to the machine.
- **/home:** Like My Documents in Windows.
- **/root:** The home directory for user root. This is usually not accessible to other users on the system.
- **/lib:** Essential shared libraries and kernel modules.
- **/dev:** Device files. These are special virtual files that help the user interface with the various devices on the system.
- **/tmp:** Temporary files. As the name suggests, programs running often store temporary files in here.
- **/boot:** Files used by the bootstrap loader. Kernel images are often kept here instead of in the root directory. If there are many kernel images, the directory can easily grow too large, and it might be better to keep it in a separate filesystem.
- **/mnt:** Mount point for mounting a filesystem temporarily.
- **/opt:** Add-on application software packages.
- **/usr:** Secondary hierarchy.
- **/var:** Variable data.
- **/sys:** Exports information about devices and drivers from the kernel device model to user space, and it is also used for configuration.
- **/proc:** Represent the current state of the kernel.

Linux boot process

These are the main stages of the Linux boot process for the BCM2837 SoC:

1. The boot process begins at POR (Power On Reset), where the hardware reset turns ON the VideoCore GPU (CPU is OFF), which executes the 1st stage bootloader from the on-chip boot ROM. At this point, DRAM memory controller is disabled.
2. The 1st stage bootloader reads the 2nd stage bootloader (bootcode.bin) from the SD card, loads it into the GPU's L2 cache and executes it on the GPU. The 2nd stage bootloader sets up the DRAM memory controller, then loads the GPU firmware (start.elf) from the SD card and runs it. The GPU firmware reads the system configuration parameters (config.txt), loads a kernel image (kernel7.img) and a Device Tree file (bcm2710-rpi-3-b.dtb) into the DRAM memory, and releases the reset signal on the ARM CPU.
3. The image kernel7.img is then run on the ARM CPU.

4. The kernel runs low level kernel initialization, enabling MMU, creating the initial table of memory pages and setting up caches. This is done in `arch/arm/kernel/head.s`. The file `head.s` contains CPU architecture specific, but platform independent initialization code. Then, the system switches to the non architecture specific kernel startup function `start_kernel()`.
5. The kernel runs `start_kernel()` located in `init/main.c` that:
 - Initializes the kernel core (e.g., memory, scheduling, interrupts).
 - Initializes statically compiled drivers.
 - Mounts the root filesystem based on bootargs passed to the kernel from U-Boot.
 - Executes the first user process, `init`. The `init` process, by default, is `/init` for `initramfs` and `/sbin/init` for a regular filesystem. The three `init` programs that you usually find on embedded Linux devices are **BusyBox init**, **System V init** and **systemd**.

Building an embedded Linux system for the Raspberry Pi

Building an embedded Linux system requires you to:

1. Select a **cross toolchain**. The toolchain is the starting point for the development process, as it is used to build all subsequent software packages. The toolchain consists of the following parts: Assembler, Compiler, Linker, Debugger, Runtime Libraries and Utilities. A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
2. Select the different packages that will run on the target (bootloader, kernel and root filesystem).
3. Configure and build these packages.
4. Deploy them on the device.

There are several different ways to build an embedded Linux system:

1. **Manually** (creating your own scripts): This option gives you total control, but it is also tedious and harder to reproduce builds on other machines. It also requires a good understanding of the software component installation process. For example, create a root filesystem from the ground up by yourself means:
 - Download the source code of all software components (libraries, utilities and applications).
 - Solve all dependencies and version conflicts, and apply patches.
 - Configure each software component.
 - Cross-compile each software component.

- Install each software component.
2. Using **complete distributions** (e.g., Ubuntu/Debian): Easy to get started, but harder to customize. A Linux distribution is a preselected kernel version and a root filesystem with a preselected set of libraries, utilities and applications.
 3. Using **build frameworks** (e.g., Buildroot, Yocto): This option allows you to customize and reproduce builds easily. This is becoming the most popular option in the embedded Linux space. A Build framework typically consists of scripts and configuration meta-data that control the build process. The Build framework typically downloads, configures, compiles and installs all required components of the system, taking version conflicts and dependencies into account. It allows, for example to create a customized root filesystem. The Build framework output is a complete image, including toolchain, bootloader, kernel and root filesystem.

Raspberry Pi OS

Raspberry Pi OS is the recommended operating system for normal use on a Raspberry Pi. Raspberry Pi OS is a free operating system based on Debian, optimised for the Raspberry Pi hardware. Raspberry Pi OS comes with over 35,000 packages: precompiled software bundled in a nice format for easy installation on your Raspberry Pi. Raspberry Pi OS is a community project under active development, with an emphasis on improving the stability and performance of as many Debian packages as possible.

You will install on a Micro SD a **Raspberry Pi OS** image based on **kernel 5.4.y**. Go to https://downloads.raspberrypi.org/raspbios_full_armhf/images/raspbios_full_armhf-2020-12-04/ and download the 2020-12-02-raspbios-buster-armhf-full.zip image.

To write the compressed image on the Micro SD card, you will download and install **Etcher**. This tool, which is an Open Source software, is useful since it allows to get a compressed image as input. More information and extra help is available on the Etcher website at <https://etcher.io/>.

Once the image is installed on the Micro SD card, you will insert the Micro SD into the SD card reader on your host PC, then you will enable UART, SPI and I2C peripherals in the programmed Micro SD. Open a terminal application on your host PC, and type the following commands:

```
~$ lsblk
~$ mkdir ~/mnt
~$ mkdir ~/mnt/fat32
~$ mkdir ~/mnt/ext4
~$ sudo mount /dev/mmcblk0p1 ~/mnt/fat32
```

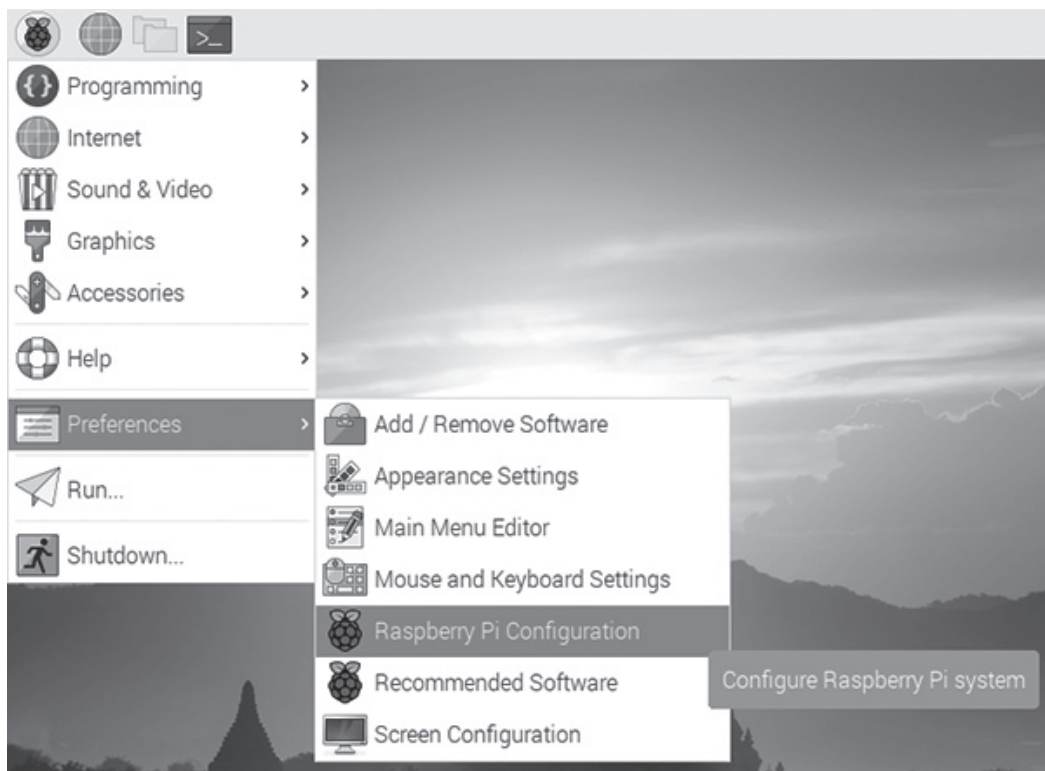
See the files in the fat32 partition. Check that config.txt is included:

```
~$ ls -l ~/mnt/fat32/
```

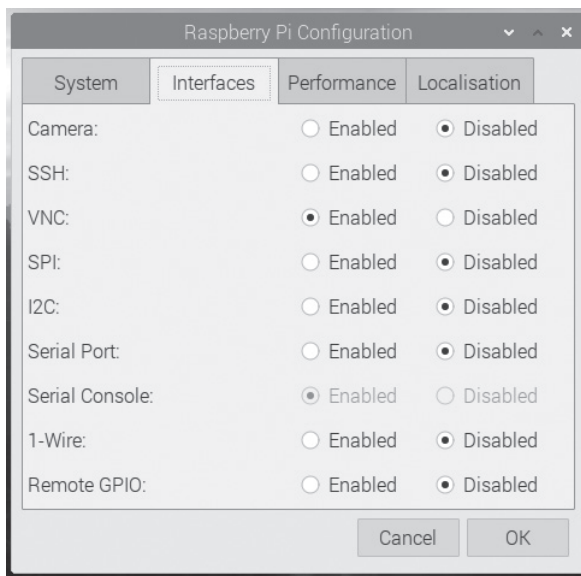

Update the config.txt file, adding the next values:

```
~$ cd ~/mnt/fat32/  
~/mnt/fat32$ sudo nano config.txt  
dtparam=i2c_arm=on  
dtparam=spi=on  
dtoverlay=spi0-cs  
# Enable UART  
enable_uart=1  
kernel=kernel7.img  
~$ sudo umount ~/mnt/fat32
```

You can also update previous settings (after booting the Raspberry Pi board) through the Raspberry Pi Configuration application found in Preferences on the menu.



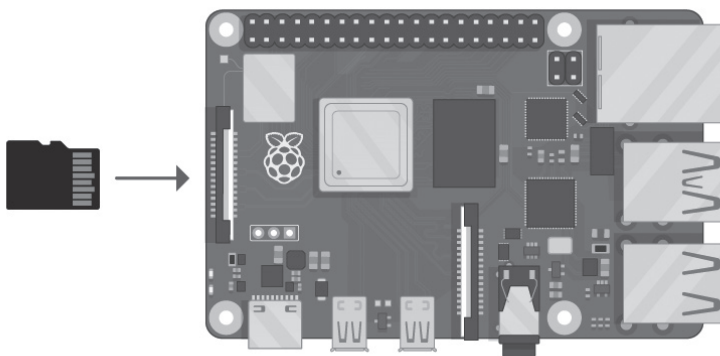
The Interfaces tab is where you turn these different connections on or off so that the Pi recognizes that you've linked something to it via a particular type of connection.



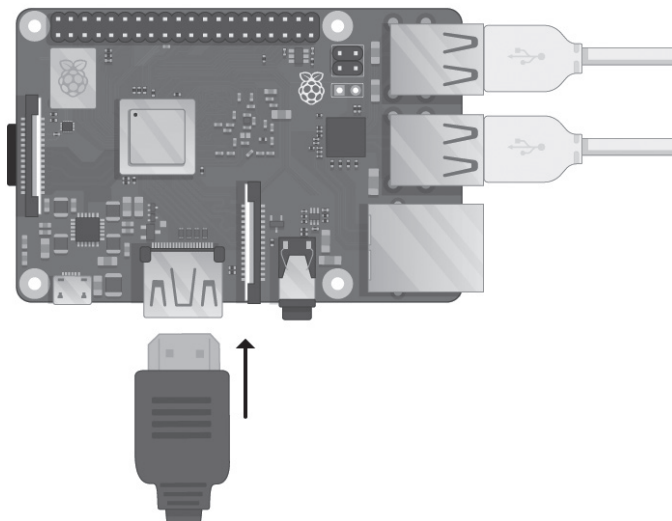
Connect and set up hardware

Follow the next steps to get everything connected to your Raspberry Pi. It's important to do this in the right order so that all your components are safe:

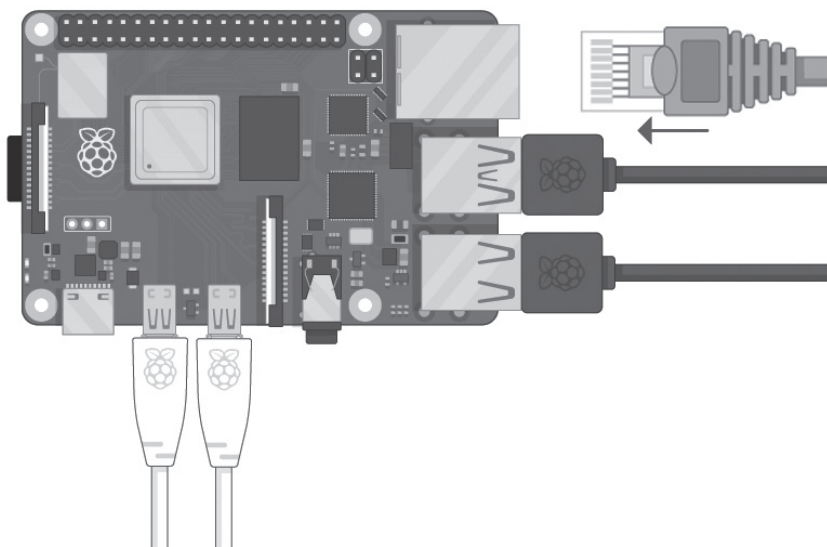
1. Insert the Micro SD card you've set up with Raspberry Pi OS into the Micro SD card slot on the underside of your Raspberry Pi board.



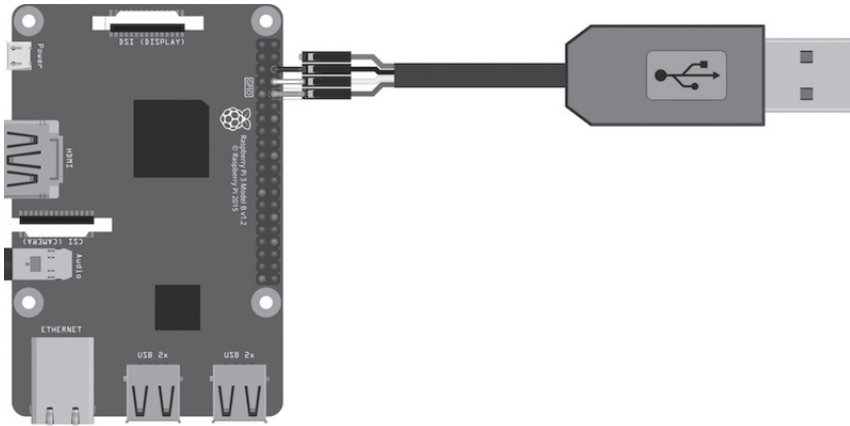
2. Connect your screen to the single Raspberry Pi's HDMI port. You can also connect a mouse to a USB port and keyboard in the same way.



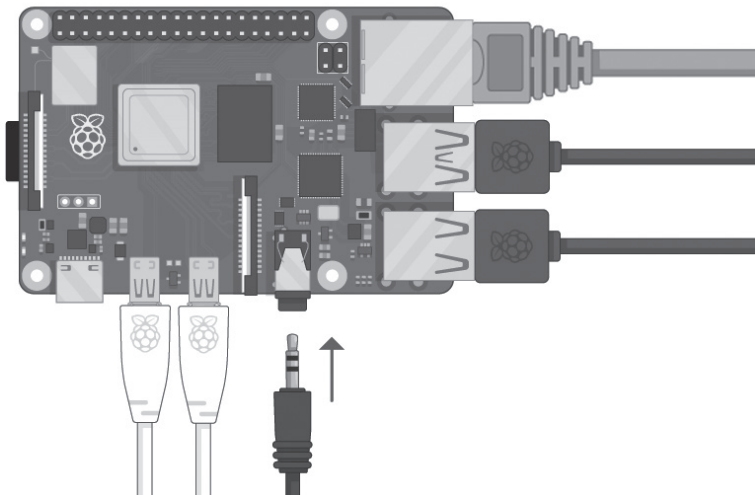
3. Connect the Ethernet port on Raspberry Pi to an Ethernet socket on your host PC.



4. The serial console is a helpful tool for debugging your board and reviewing system log information. To access the serial console, connect a USB to TTL Serial Cable to the device UART pins as shown below.



5. Plug the USB power supply into a socket and connect it to your Raspberry Pi's power port.



The hardware setup is already done!!

You should see a red LED light up, which indicates that the Raspberry Pi board is connected to power. As it starts up, you will see raspberries appear in the top left-hand corner of your screen. After a few seconds the Raspberry Pi OS Desktop will appear.



For the official Raspberry Pi OS, the default user name is **pi**, with password **raspberrypi**.

To find out the version of the booted kernel, run `uname -r` on the Raspberry Pi terminal:

```
pi@raspberrypi:~$ uname -r
5.4.79-v7+
```

Reset the board. You can disconnect your screen from the Raspberry Pi's HDMI port during the development of the labs (you will only need to connect a screen in the LAB 10.3).

```
pi@raspberrypi:~$ sudo reboot
```

To see Linux boot messages on the console, add `loglevel=8` in the file `cmdline.txt` under `/boot`.

```
pi@raspberrypi:~$ sudo nano /boot/cmdline.txt
```

To change your current `console_loglevel` simply write to this file:

```
pi@raspberrypi:~$ echo <loglevel> > /proc/sys/kernel/printk
```

For example:

```
pi@raspberrypi:~$ echo 8 > /proc/sys/kernel/printk
```

Now, every kernel message will appear on your console, as all priority higher than 8 (lower loglevel values) will be displayed. Please note that after reboot, this configuration is reset. To keep the

configuration permanently, just append the following `kernel.printk` value to the `/etc/sysctl.conf` file, then reboot the processor:

```
kernel.printk = 8 4 1 3
pi@raspberrypi:~$ sudo nano /etc/sysctl.conf
pi@raspberrypi:~$ sudo reboot
```

Setting up an ethernet communication

Connect an Ethernet cable between your host PC and the Raspberry Pi board. Set up the IP Address of the host PC.

1. On the host side, click on the Network Manager tasklet on your desktop, and select Edit Connections. Choose "Wired connection 1" and click "Edit".
2. Choose the "IPv4 Settings" tab, and select Method as "Manual" to make the interface use a static IP address, for example 10.0.0.1. Click "Add", and set the IP address, the Netmask and Gateway as follow:

Address: 10.0.0.1

Netmask: 255.255.255.0

Gateway: none or 0.0.0.0

Finally, click the "Save" button.

3. Click on "Wired connection 1" to activate this network interface.

Copying files to your Raspberry Pi

You can access the command line of a Raspberry Pi remotely from another computer or device on the same network using SSH. Make sure the Raspberry Pi is properly set up and connected. Configure on the Raspberry Pi the `eth0` interface with IP address 10.0.0.10:

```
pi@raspberrypi:~$ sudo ifconfig eth0 10.0.0.10 netmask 255.255.255.0
```

Raspbian has the SSH server disabled by default. You have to start the service on the Pi:

```
pi@raspberrypi:~# sudo /etc/init.d/ssh restart
```

Now, verify that you can ping your Linux host machine from the Raspberry Pi. Exit the ping command by typing "Ctrl-c":

```
pi@raspberrypi:~# ping 10.0.0.1
```

You can also ping from Linux host machine to the target. Exit the ping command by typing "Ctrl-c".

```
~$ ping 10.0.0.10
```

By default, the root account is disabled, but you can enable it by using this command and giving it a password, for example "pi":

```
pi@raspberrypi:~$ sudo passwd root
```

Now, you can log into your pi as the root user. Open the `sshd_config` file, and change **PermitRootLogin** to **yes** (also comment the line out). After editing the file, type "Ctrl+x", then type "yes", and press "enter" to exit:

```
pi@raspberrypi:~$ sudo nano /etc/ssh/sshd_config
```

Building the Linux kernel

There are two main methods for building the kernel. You can build locally on the Raspberry Pi, which will take a long time, or you can cross-compile, which is much quicker, but requires more setup. You will use the second method.

Install Git and the build dependencies:

```
~$ sudo apt install git bc bison flex libssl-dev make libc6-dev libncurses5-dev
```

On your host PC, create the `linux_rpi3` folder, where you are going to download the kernel sources:

```
~$ mkdir linux_rpi3
~$ cd linux_rpi3/
```

Get the kernel sources. The git clone command below will download the current active branch without any history. Omitting the `--depth=1` will download the entire repository, including the full history of all branches, but this takes much longer and occupies much more storage.

```
~/linux_rpi3$ git clone --depth=1 -b rpi-5.4.y https://github.com/raspberrypi/linux
```

Install the 32-bit toolchain for a 32-bit kernel:

```
~$ sudo apt install crossbuild-essential-armhf
```

Compile the kernel, modules and Device Tree files. First, apply the default configuration:

```
~/linux_rpi3/linux$ KERNEL=kernel7
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
```

Configure the following kernel settings that will be needed during the development of the labs:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
Device drivers >
```

```
  [*] SPI support --->
      <*> User mode SPI device driver support
```

```
Device drivers >
  <*> Industrial I/O support --->
      *- Enable buffer support within IIO
```

```

    *- Industrial I/O buffering based on kfifo
    <*> Enable IIIO configuration via configs
    *- Enable triggered sampling support
    <*> Enable software IIIO device support
    <*> Enable software triggers support
        Triggers - standalone --->
            <*> High resolution timer trigger
            <*> SYSFS trigger

Device drivers >
    <*> Userspace I/O drivers --->
        <*> Userspace I/O platform driver with generic IRQ handling

Device drivers >
    Input device support --->
        *- Generic input layer (needed for keyboard, mouse, ...)
        <*> Polled input device skeleton

```

For the LAB 12.1, you will need the functions that enable the triggered buffer support. If they are not defined accidentally by another driver, there's an error thrown out while linking. To solve this problem, you can select, for example, the HTS221 driver, which includes this triggered buffer support:

```

Device drivers >
    <*> Industrial I/O support > Humidity sensors --->
        <*> STMicroelectronics HTS221 sensor Driver

```

Save the configuration, and exit from menuconfig.

Compile kernel, Device Tree files and modules in a single step:

```
~/linux_rpi3/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- zImage modules dtbs
```

Having built the kernel, you need to copy it onto your Raspberry Pi device and also install the modules. Insert the previously programmed Micro SD into the SD card reader on your host PC, and execute the following commands:

```

~$ lsblk
~$ mkdir ~/mnt
~$ mkdir ~/mnt/fat32
~$ mkdir ~/mnt/ext4
~$ sudo mount /dev/mmcblk0p1 ~/mnt/fat32
~$ sudo mount /dev/mmcblk0p2 ~/mnt/ext4/
~$ cd linux_rpi3/linux
~/linux_rpi3/linux$ sudo env PATH=$PATH make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
INSTALL_MOD_PATH=~/mnt/ext4 modules_install

```


Finally, update kernel, Device Tree files and modules:

```
~/linux_rpi3/linux$ sudo cp ~/mnt/fat32/kernel7.img ~/mnt/fat32/kernel7-backup.img
~/linux_rpi3/linux$ sudo cp arch/arm/boot/zImage ~/mnt/fat32/kernel7.img
~/linux_rpi3/linux$ sudo cp arch/arm/boot/dts/*.dtb ~/mnt/fat32/
~/linux_rpi3/linux$ sudo cp arch/arm/boot/dts/overlays/*.dtb* ~/mnt/fat32/overlays/
~/linux_rpi3/linux$ sudo cp arch/arm/boot/dts/overlays/README ~/mnt/fat32/overlays/
~$ sudo umount ~/mnt/fat32
~$ sudo umount ~/mnt/ext4
```

Insert the Micro SD card you've set up in the Micro SD card slot on the underside of your Raspberry Pi, and connect the Raspberry Pi's UART (through a USB to serial adapter) to a Linux PC's USB. On Linux PC, launch Minicom utility as shown below (for debugging purpose):

```
~$ sudo minicom -s
```

Set baud rate and other setting as per below:

1. Baud rate 115200
2. Parity none
3. Hardware flow control/software flow control none
4. Serial device /dev/ttyUSB0
5. Parity none

After the Raspberry Pi board boots up, it will display a login console on Minicom terminal on Linux. The username for the board is "pi" and password is "raspberrypi".

To find out the version of your new kernel, boot the system and run `uname -r`:

```
pi@raspberrypi:~$ uname -r
5.4.83-v7+
```

If you modify and compile the kernel or Device Tree files later, you can copy them to the Raspberry Pi remotely using the secure copy protocol (SCP). You need to connect previously an Ethernet cable between the Raspberry Pi board and your host PC.

```
~/linux_rpi3/linux$ scp arch/arm/boot/zImage root@10.0.0.10:/boot/kernel7.img
```

Copy the following .dtb file if you are using the Raspberry Pi 3 Model B board:

```
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

Copy the following .dtb file if you are using the Raspberry Pi 3 Model B+ board:

```
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b-plus.dtb root@10.0.0.10:/boot/
```


2

The Linux Device and Driver Model

Understanding the Linux device and driver model is central to developing device drivers in Linux. A unified device model was added in Linux kernel 2.6 to provide a single mechanism for representing devices and describing their topology in the system. The Linux device and driver model is a universal way of organizing devices and drivers into buses. Such a system provides several benefits:

- Minimization of code duplication.
- Clean code organization with the device drivers separated from the controller drivers, the hardware description separated from the drivers themselves, etc.
- Capability to determine all the devices in the system, view their status and power state, see what bus they are attached to, and determine which driver is responsible for them.
- The capability to generate a complete and valid tree of the entire device structure of the system, including all buses and interconnections.
- The capability to link devices to their drivers and vice versa.
- Categorization of devices by their type (classes), such as input devices, without the need to understand the physical device topology.

The device model involves terms like "device", "driver" and "bus":

- **device:** A physical or virtual object which attaches to a bus.
- **driver:** A software entity which may probe for and be bound to devices and which can perform certain management functions.
- **bus:** A device which serves as an attachment point for other devices.

The device model is organized around three main data structures:

1. The `bus_type` structure, which represents one type of bus (e.g., USB, PCI, I2C).
2. The `device_driver` structure, which represents one driver capable of handling certain devices on a certain bus.
3. The device structure, which represents one device connected to a bus.

Bus core drivers

For each bus supported by the kernel, there is a generic bus core driver. A bus is a channel between the processor and one or more devices. For the purposes of the device model, all devices are connected via a bus, even if it is an internal, virtual, "platform" bus.

The bus core driver allocates a `bus_type` structure and registers it with the kernel's list of bus types. The `bus_type` structure (declared in `include/linux/device.h` in the kernel source tree) represents one type of bus (USB, PCI, I2C, etc.). The registration of the bus in a system is done by using the `bus_register()` function. The `bus_type` structure is declared as follows:

```
struct bus_type {
    const char *name;
    const char *dev_name;
    struct device *dev_root;
    struct device_attribute *dev_attrs;
    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 (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);

    int (*online)(struct device *dev);
    int (*offline)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    const struct dev_pm_ops *pm;

    struct iommu_ops *iommu_ops;

    struct subsys_private *p;
    struct lock_class_key lock_key;
};
```

A new bus must be registered via a call to `bus_register()`, which is defined in `drivers/base/platform.c` in the kernel source tree:

```
struct bus_type platform_bus_type = {
    .name           = "platform",
    .dev_groups     = platform_dev_groups,
    .match          = platform_match,
    .uevent         = platform_uevent,
    .pm             = &platform_dev_pm_ops,
};
EXPORT_SYMBOL_GPL(platform_bus_type);
```

```

int __init platform_bus_init(void)
{
    int error;

    early_platform_cleanup();

    error = device_register(&platform_bus);
    if (error)
        return error;
    error = bus_register(&platform_bus_type);
    if (error)
        device_unregister(&platform_bus);

    return error;
}

```

The `bus_register()` function registers the bus with the kobject infrastructure and creates a `/sys/bus/platform/` directory that consists of two directories: `devices` and `drivers`.

One of the `struct bus_type` members is a pointer to the `subsys_private` structure, which is declared in `drivers/base/base.h` in the kernel source tree:

```

struct subsys_private {
    struct kset subsys;
    struct kset *devices_kset;
    struct list_head interfaces;
    struct mutex mutex;

    struct kset *drivers_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;

    struct kset glue_dirs;
    struct class *class;
};

```

The `klist_devices` member of the `subsys_private` structure is a list of devices in the system that reside on this particular type of bus. This list is updated by the `device_register()` function, which is called when the bus is scanned for devices by the bus controller driver (during initialization or when a device is hot plugged).

The `klist_drivers` member of the `subsys_private` structure is a list of drivers that can handle devices on that bus. This list is updated by the `driver_register()` function, which is called when a driver initializes itself.

When a new device is plugged into the system, the bus controller driver detects the device and calls `device_register()`. When a device is registered by the bus controller driver, the parent member

of the device structure is pointed to the bus controller device to build the physical device list. The list of drivers associated with the bus is iterated over to find out if there are any drivers that can handle the device. The match function provided in the `bus_type` structure is used to check if a given driver can handle a given device. When a driver is found that can handle the device, the driver member of the device structure is pointed to the corresponding device driver.

When a kernel module is inserted into the kernel and the driver calls `driver_register()`, the list of devices associated with the bus is iterated over to find out if there are any devices that the driver can handle by using the match function. When a match is found, the device is associated with the device driver, and the driver's `probe()` function is called. This is what we call **binding**.

When does a driver attempt to bind a device?

1. When the driver is registered (if the device already exists).
2. When the device is created (if the driver is already registered).

Summarizing, the bus driver registers a bus in a system and:

1. Allows registration of bus controller drivers, whose role is to detect devices and configure their resources.
2. Allows registration of device drivers.
3. Matches devices and drivers.

Bus controller drivers

For a specific bus type, there could be many different controllers provided by different vendors. Each of these controllers needs a corresponding bus controller driver. The role of a bus controller driver, in the maintenance of the device model, is similar to that of any other device driver, in that it registers itself to its bus by using the `driver_register()` function. In most cases, these bus controller devices are autonomous entities in the system, discovered during the kernel initialization calling `of_platform_populate()`, which walks through the DT, finding and registering these "platform controller devices" to the platform bus at runtime.

Device drivers

Every device driver registers itself with the bus core driver by using `driver_register()`. After that, the device model core tries to bind it with a device. When a device that can be handled by a particular driver is detected, the `probe()` member of the driver is called, and the device configuration data can be retrieved from the Device Tree.

Each device driver is responsible for instantiating and registering an instance of the `device_driver` structure (declared in `include/linux/device.h` in the kernel source tree) with the device model core. The `device_driver` structure is declared as follows:

```
struct device_driver {
    const char *name;
    struct bus_type *bus;

    struct module *owner;
    const char *mod_name;

    bool suppress_bind_attrs;

    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 driver_private *p;
};
```

See below the description of some of the main members of `struct device_driver`:

- The `bus` member is a pointer to the `bus_type` structure to which the device driver is registered.
- The `probe` member is a callback function that is called for each device detected that is supported by the driver. The driver should instantiate itself for each device and initialize the device as well.
- The `remove` member is a callback function that is called to unbind the driver from the device. This happens when the device is physically removed, when the driver is unloaded, or when the system is shut down.

Devices

At the lowest level, every device in a Linux system is represented by an instance of `struct device`. The device structure contains the information that the device model core needs to model the system. Most subsystems, however, track additional information about the devices they host. As a result, it is rare for devices to be represented by bare device structures; instead, that structure, like the `kobject` structure, is usually embedded within a higher-level representation of the device. For example, a `struct device` is included inside the definition of the `platform_device` structure. The `struct platform_device` is declared as follows:

```

struct platform_device {
    const char    *name;
    int           id;
    bool          id_auto;
    struct device dev;
    u64           platform_dma_mask;
    u32           num_resources;
    struct resource *resource;

    const struct platform_device_id *id_entry;
    char *driver_override; /* Driver name to force a match */

    /* MFD cell pointer */
    struct mfd_cell *mfd_cell;

    /* arch specific additions */
    struct pdev_archdata archdata;
};

```

You can see below a code snippet of the device structure, which is declared in `include/linux/device.h` in the kernel source tree:

```

struct device {
    struct kobject kobj;
    struct device *parent;
    struct device_private *p;
    const char *init_name; /* initial name of the device */
    const struct device_type *type;
    struct bus_type *bus; /* type of bus device is on */
    struct device_driver *driver; /* which driver has allocated this
                                   device */
    void *platform_data; /* Platform specific data, device
                           core doesn't touch it */
    void *driver_data; /* Driver data, set and get with
                        dev_set_drvdata/dev_get_drvdata */

    [...]

    struct device_node *of_node; /* associated device tree node */
    struct fwnode_handle *fwnode; /* firmware device node */

    [...]

    struct class *class;
    const struct attribute_group **groups; /* optional groups */
    void (*release)(struct device *dev);

    [...]
};

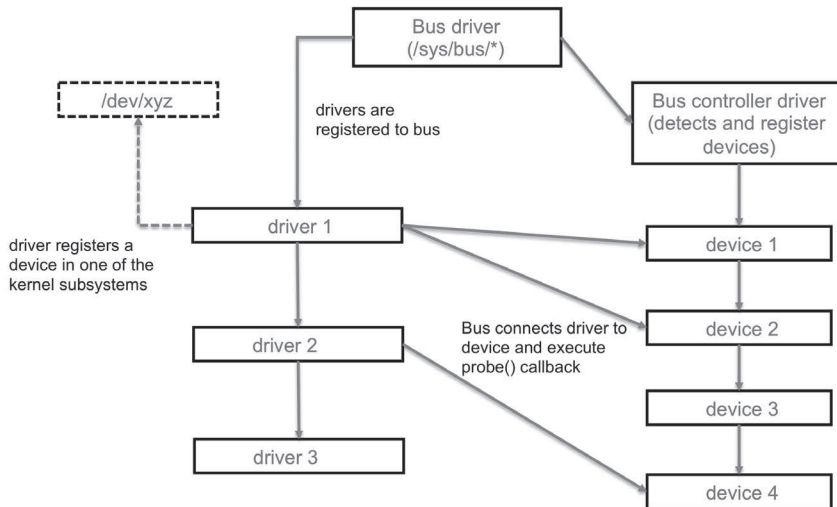
```

Following is a brief description of a few important members of the device structure:

- **kobj**: The kobject that represents the device and links it into the hierarchy.

- **parent**: The device's "parent" device, the device to which it is attached. In most cases, a parent device is some sort of bus or host controller. If the parent is NULL, the device is a top-level device, which is not usually what you want.
- **type**: The type of device. This identifies the device type and carries type-specific information.
- **bus**: Type of bus, device is on.
- **driver**: Which driver has allocated this.
- **driver_data**: Private pointer for driver specific info that may be used by the device driver.
- **release()**: Callback to free the device after all references have gone away. This should be set by the allocator of the device (i.e., the bus driver that discovered the device). This is called from the embedded kobject `release()` method.

The Linux device model is illustrated in the following figure:



The sysfs filesystem

Sysfs is a virtual filesystem that exports information about devices and drivers from the kernel device model to user space. It provides a means to export kernel data structures, their attributes, and the linkages between them with user space. Various programs use sysfs: `udev`, `mdev`, `lsusb`.

Since the Linux device driver model was introduced in version 2.6, the `sysfs` represents all devices and drivers as kernel objects. You can see the kernel's view of the system laid by looking under `/sys/`, as shown here:

- `/sys/bus/` -- Contains one subdirectory for each of the bus types in the kernel.
- `/sys/devices/` -- Contains the list of devices.
- `/sys/bus/<bus>/devices/` -- Devices on a given bus.
- `/sys/bus/<bus>/drivers/` -- Drivers on a given bus.
- `/sys/class/` -- This subdirectory contains a single layer of further subdirectories for each of the device classes that have been registered on the system (for example, terminals, network devices, block devices, graphics devices, sound devices, and so on). Inside each of these subdirectories are symbolic links for each of the devices in this class. These symbolic links refer to entries in the `/sys/devices/` directory.
- `/sys/bus/<bus>/devices/<device>/driver/` -- Symlink to driver that manages the given device.

Let's focus now on two of the directories shown above:

1. **The list of devices:** `/sys/devices/`:

This directory contains a filesystem representation of the Device Tree. It maps directly to the internal Device Tree, which is a hierarchy of the device structures. There are three directories that are present on all systems:

- **system:** This contains devices at the heart of the system, including a collection of both global and individual CPU attributes and clocks.
- **virtual:** This contains devices that are memory-based. You will find the memory devices that appear as `/dev/null`, `/dev/random` and `/dev/zero` in `virtual/mem`. You will find the loopback device, `lo`, in `virtual/net`.
- **platform:** This contains devices that are not connected via a conventional hardware bus. This could be almost anything on an embedded device.

Devices will be added and removed dynamically as the machine runs, and between different kernel versions, the layout of the devices within this tree will change. Do not rely on the format of this tree because of this. If a program wishes to find different things in the tree, use the `/sys/class/` structure and rely on the symlinks there to point to the proper location within the `/sys/devices/` tree of the individual devices.

2. **The device drivers grouped by classes:** `/sys/class/`:

The `/sys/class/` directory consist of a group of subdirectories describing individual classes of devices in the kernel. The individual directories consist of either subdirectories or symlinks to other directories.

For example, you will find I2C controller drivers under `/sys/class/i2c-dev/`. Each registered I2C adapter gets a number, starting from 0. You can examine `/sys/class/i2c-dev/` to see which number corresponds to which adapter.

Some attribute files are writable and allow you to tune parameters in the driver at runtime. The `dev` attribute is particularly interesting. If you look at its value, you will find the major and minor numbers of the device.

The kobject infrastructure

Sysfs is tied inherently to the **kobject** infrastructure. The kobject is the fundamental structure of the device model, and it allows sysfs representation. The kobject structure is declared as follows:

```
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;

    [...]
}
```

Following is a brief description of few important fields of the kobject structure:

- **name**: Name of the kobject. The current kobject is created with this name in sysfs.
- **parent**: This is the kobject's parent. The kobject directory will be found in its parent directory, as determined by the kobject's parent field.
- **ktype**: The type associated with a kobject.
- **kset**: A group of kobjects all of which are embedded in structures of the same type.
- **sd**: Points to a `sysfs_dirent` structure that represents this kobject in sysfs.
- **kref**: Provides reference counting.

The kobject structures are in a hierarchy: an object has a parent and holds a `kset` member, which contains objects on the same level. A kobject structure does not perform a single function. This structure is usually integrated into a larger one. A kobject structure actually incorporates a set of features that will be offered to a higher abstraction object in the Linux Device Model hierarchy. For example, the `cdev` structure is declared as follows:

```
struct cdev {
    struct kobject kob;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
```

```
    unsigned int count;
};
```

The `kobj` field of `struct cdev` is initialized by calling `cdev_alloc()` and named in `register_chrdev()`.

For every `kobject` that is registered with the system, a directory is created for it in the `sysfs`. That directory is created as a subdirectory of the `kobject`'s parent, expressing internal object hierarchies to user space. Every `kobject` needs to have an associated `kobj_type` structure. The `Kobj_type` structure controls what happens when the `kobject` is created and deleted. The pointer to this structure can be found in two different places:

1. Inside the `kobject` structure, there is a pointer `Ktype` to the `kobj_type` structure.
2. If, however, this `kobject` is a member of a `kset`, the `kobj_type` pointer is provided by that `kset` instead. A `kset` is a collection of `kobjects` embedded within structures of the same type. For example, a `kset` can be used by the kernel to track "all block devices" or "all PCI device drivers".

The `object_type` structure is declared as follows:

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

The `Kobj_type` structure also contains attributes that can be exported to `sysfs` directories (directories are created with `kobject_add()` as files (files are created with `sysfs_create_file()`). All `kobjects` of the same type have the same default set of files populating their `sysfs` directories. The `kobj_type` structure contains a member, `default_attrs`, that is an array of attribute structures. These attributes can be "read/write" via "show/store" methods by using the `sysfs_ops` structure that is pointed via the `sysfs_ops` pointer variable (included in `struct kobj_type`). The `sysfs_ops` structure is declared as follows:

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *, struct attribute *, char *);
    ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
};
```

An attribute definition is shown below:

```
struct attribute {
    char *name;
    struct module *owner;
    umode_t mode;
};
```

The default attributes provided by the ktype associated with a kobject, sometimes are not sufficient. Subsystems are encouraged to define their own attribute structure and wrapper functions for adding and removing attributes for a specific object type. For example, the kernel device subsystem declares struct device_attribute like:

```
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);
};
```

It also defines this helper for defining device attributes:

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
    struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

For example, declaring:

```
static DEVICE_ATTR(foo, S_IWUSR | S_IRUGO, show_foo, store_foo);
```

is equivalent to doing:

```
static struct device_attribute dev_attr_foo = {
    .attr = {
        .name = "foo",
        .mode = S_IWUSR | S_IRUGO,
    },
    .show = show_foo,
    .store = store_foo,
};
```

The macro DEVICE_ATTR requires the following inputs:

- A name for the attribute in the filesystem.
- Permissions which determine if the attribute can be read and/or written. The macros for modes are defined in include/linux/stat.h.
- A function to read the data from the driver.
- A function to write the data into the driver.

You can add/remove a "sysfs attribute file" for a device by using the following 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 device_create_file() function (defined in drivers/base/core.c in the kernel source tree) calls sysfs_create_file(), which adds new attributes on top of the default set. The device_remove_file() function is defined as follows:

```
/*
 * device_create_file - create sysfs attribute file for device.
 * @dev: device.
```

```

* @attr: device attribute descriptor.
*/
int device_create_file(struct device *dev, const struct device_attribute *attr)
{
    int error = 0;

    if (dev) {
        WARN(((attr->attr.mode & S_IWUGO) && !attr->store),
             "Attribute %s: write permission without 'store'\n",
             attr->attr.name);
        WARN(((attr->attr.mode & S_IRUGO) && !attr->show),
             "Attribute %s: read permission without 'show'\n",
             attr->attr.name);
        error = sysfs_create_file(&dev->kobj, &attr->attr);
    }

    return error;
}

```

For example, you can create two structures of type `device_attribute` with respective names **foo1** and **foo2**:

```

static DEVICE_ATTR(foo1, S_IWUSR | S_IRUGO, show_foo1, store_foo1);
static DEVICE_ATTR(foo2, S_IWUSR | S_IRUGO, show_foo2, store_foo2);

```

These two attributes can be organized as follows into a group:

```

static struct attribute *dev_attrs[] = {
    &dev_attr_foo1.attr,
    &dev_attr_foo2.attr,
    NULL,
};

static struct attribute_group dev_attr_group = {
    .attrs = dev_attrs,
};

static const struct attribute_group *dev_attr_groups[] = {
    &dev_attr_group,
    NULL,
};

```

You can add/remove a group of "sysfs attribute files" by using the next functions:

```

int sysfs_create_group(struct kobject *kobj, const struct attribute_group *grp);
void sysfs_remove_group(struct kobject * kobj, const struct attribute_group * grp);

```

You can add/remove a group of "sysfs attribute files, for example, to an I2C client device:

```

int sysfs_create_group(&client->dev.kobj, &dev_attr_group);
void sysfs_remove_group(&client->dev.kobj, &dev_attr_group);

```

Introduction to the Device Tree

The Device Tree specification (DTSpec) has its origin in the IEEE 1275 specification, which was created to address problems of general purpose computers, such as how a single version of an operating system can work on several different computers within the same family. The Device Tree specification was born more oriented to work with embedded systems, omitting some importing features of the IEEE 1275 specification and retaining the boot program to client program interface definition, by which a program can describe and communicate system hardware information to a client program, thus eliminating the need for the client program to have hard-coded descriptions of system hardware.

DTSpec specifies a construct called a **Device Tree** (DT) to describe system hardware (a SoC and its target boards). Rather than hard coding every detail of every SoC and its target boards into the Linux kernel, many aspects of the hardware can be described in a data structure that is passed to the operating system at boot time. In earlier kernel versions, this information was stored in several C files in the Linux kernel itself. Due to the growing number of new ARM SoCs, the amount of code grew too fast. The Device Tree solves this. An extract of the DTSpec defines Device Tree as follows:

A devicetree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent.

You can download the full DTSpec at <https://github.com/devicetree-org/devicetree-specification/releases/>, where you can find a detailed description of the Device Tree structure and conventions.

Conceptually, a common set of usage conventions, called "bindings", is defined for how data should appear in the Device Tree to describe typical hardware characteristics of a new device. A binding should be created that fully describes the required properties of the device. This set of properties shall be sufficiently descriptive to provide Linux device drivers with needed attributes of the device. In the case of an embedded processor, these properties would include data busses, interrupt lines, GPIO connections and peripheral devices. As much as possible, hardware is described using existing bindings to maximize use of existing support code, but since property and node names are text strings, it is easy to extend existing bindings or create new ones by defining new nodes and properties. Bindings are described in the DTSpec for generic bindings and in the Linux software component documentations, such as the Linux kernel: <https://www.kernel.org/doc/Documentation/devicetree/bindings/> and U-Boot: <https://github.com/ARM-software/u-boot/tree/master/doc/device-tree-bindings>.

In the Device Tree, the compatible property is of great importance during the development of Linux device drivers. Its value consists of one or more strings that define the specific programming model for the device. The compatible property value is formed by a concatenated list of null terminated strings, from most specific to most general.

The DT is represented as a set of text files in the Linux kernel source tree. They are located under `arch/arm/boot/dts/` and can have two extensions:

- ***.dtsi** files are Device Tree source include files. They describe the hardware that is common to several platforms which include these files on their *.dts files.
- ***.dts** files are Device Tree source files. They describe one specific platform.

Linux uses DT data for three major purposes:

1. **Platform Identification:** The kernel will use data in the DT to identify the specific machine. In a perfect world, the specific platform shouldn't matter to the kernel because all platform details would be described perfectly by the Device Tree in a consistent and reliable manner. Hardware is not perfect though, and the kernel must identify the machine during early boot so that it has the opportunity to run machine-specific fixups. In the majority of cases, the machine identity is irrelevant, and the kernel will instead select setup code based on the machine's core CPU or SoC. On ARM, for example, `setup_arch()` (defined in `arch/arm/kernel/setup.c` in the kernel source tree) calls `setup_machine_fdt()` (defined in `arch/arm/kernel/devtree.c` in the kernel source tree), which searches through the `machine_desc` table and selects the `machine_desc` which best matches the Device Tree data. It determines the best match by looking at the `compatible` property in the root Device Tree node and comparing it with the `dt_compat` list of the `machine_desc` structure, which is declared in `arch/arm/include/asm/mach/arch.h` in the kernel source tree.

In the Device Tree, the `compatible` property contains a sorted list of strings starting with the exact name of the machine. For the Broadcom BCM2837 SoC, the `bcm2837.dtsi` file (located in `arch/arm/boot/dts` in the kernel source tree) contains the following `compatible` property:

```
compatible = "brcm,bcm2837";
```

Again on ARM, for each `machine_desc`, the kernel looks to see if any of the `dt_compat` list entries appears in the `compatible` property. If one does, then that `machine_desc` is a candidate for driving the machine. For the Broadcom BCM2837 SoC, the `bcm2835_compat[]` and `DT_MACHINE_START` declarations (located in `arch/arm/mach-bcm/board_bcm2835.c` in the kernel source tree) are used to populate a `machine_desc` structure.

```
static const char * const bcm2835_compat[] = {
#ifdef CONFIG_ARCH_MULTI_V6
    "brcm,bcm2835",
#endif
#ifdef CONFIG_ARCH_MULTI_V7
    "brcm,bcm2836",
    "brcm,bcm2837",
#endif
    NULL
};
```



```

DT_MACHINE_START(BCM2835, "BCM2835")
    .dt_compat = bcm2835_compat,
    .smp = smp_ops(bcm2836_smp_ops),
MACHINE_END

```

The `setup_machine_fdt()` function calls `of_flat_dt_match_machine()`, which searches the entire table of `machine_descs`, returning the "most compatible" `machine_desc` based on which entry in the `compatible` property each `machine_desc` matches against. If no matching `machine_desc` is found, then it returns `NULL`. The `arch_get_next_mach()` function takes out a `machine_desc` and returns the `dt_compat` member in it. Each call to the `arch_get_next_mach()` function returns the `dt_compat` member of the next `machine_desc`. The `dt_compat` member which was taken out, is used to compare and match with the `compatible` property which includes the "brcm,bcm2837" string (for the BCM2837 SoC).

```

/*
 * setup_machine_fdt - Machine setup when an dtb was passed to the kernel
 * @dt: virtual address pointer to dt blob
 * If a dtb was passed to the kernel, then use it to choose the correct
 * machine_desc and to setup the system.
 */
const struct machine_desc * __init setup_machine_fdt(void *dt)
{
    const struct machine_desc *mdesc;
    unsigned long dt_root;

    if (!early_init_dt_scan(dt))
        return NULL;

    mdesc = of_flat_dt_match_machine(NULL, arch_get_next_mach);
    if (!mdesc)
        machine_halt();

    dt_root = of_get_flat_dt_root();
    arc_set_early_base_baud(dt_root);

    return mdesc;
}
/*
 * of_flat_dt_match_machine - Iterate match tables to find matching machine.
 * @default_match: A machine specific ptr to return in case of no match.
 * @get_next_compat: callback function to return next compatible match table.
 * Iterate through machine match tables to find the best match for the machine
 * compatible string in the FDT.
 */
const void * __init of_flat_dt_match_machine(const void *default_match,
    const void * (*get_next_compat)(const char * const*))
{
    const void *data = NULL;
    const void *best_data = default_match;
    const char *const *compat;

```

```

unsigned long dt_root;
unsigned int best_score = ~1, score = 0;

dt_root = of_get_flat_dt_root();
while ((data = get_next_compat(&compat))) {
    score = of_flat_dt_match(dt_root, compat);
    if (score > 0 && score < best_score) {
        best_data = data;
        best_score = score;
    }
}
if (!best_data) {
    const char *prop;
    int size;

    pr_err("\n unrecognized device tree list:\n[ ");

    prop = of_get_flat_dt_prop(dt_root, "compatible", &size);
    if (prop) {
        while (size > 0) {
            printk("%s' ", prop);
            size -= strlen(prop) + 1;
            prop += strlen(prop) + 1;
        }
        printk("]\n\n");
        return NULL;
    }

    pr_info("Machine model: %s\n", of_flat_dt_get_machine_name());

    return best_data;
}

```

2. **Runtime configuration:** In most cases, a DT will be the only method of communicating data from u-boot to the kernel, so DT can be used to pass in runtime configuration data like the kernel parameters string and the location of an initrd image. Most of this data is contained in the chosen node, and during booting Linux will look something like this:

```

chosen {
    stdout-path = "serial0:115200n8";
};

```

The chosen node above is included in the `/arch/arm/boot/dts/bcm283x.dtsi` file in the kernel source tree. The `stdout-path` property specifies the device to be used for the boot console output.

The `setup_machine_fdt()` function is also responsible for early scanning of the Device Tree. During early boot, the `setup_machine_fdt()` function calls `of_scan_flat_dt()` several times with different helper callbacks to parse the Device Tree data before paging is set up.

The `of_scan_flat_dt()` code scans through the Device Tree and uses the helpers to extract information required during early boot. Typically, the `early_init_dt_scan_chosen()` helper parses the chosen node, `early_init_dt_scan_root()` initializes the DT address space model, and `early_init_dt_scan_memory()` determines the size and location of usable RAM.

```
/*
 * setup_machine_fdt - Machine setup when an dtb was passed to the kernel
 * @dt: virtual address pointer to dt blob
 * If a dtb was passed to the kernel, then use it to choose the correct
 * machine_desc and to setup the system.
 */
const struct machine_desc * __init setup_machine_fdt(void *dt)
{
    const struct machine_desc *mdesc;
    unsigned long dt_root;

    if (!early_init_dt_scan(dt))
        return NULL;

    mdesc = of_flat_dt_match_machine(NULL, arch_get_next_mach);
    if (!mdesc)
        machine_halt();

    dt_root = of_get_flat_dt_root();
    arc_set_early_base_baud(dt_root);

    return mdesc;
}

bool __init early_init_dt_scan(void *params)
{
    bool status;

    status = early_init_dt_verify(params);
    if (!status)
        return false;

    early_init_dt_scan_nodes();
    return true;
}

void __init early_init_dt_scan_nodes(void)
{
    int rc = 0;

    /* Retrieve various information from the /chosen node */
    rc = of_scan_flat_dt(early_init_dt_scan_chosen, boot_command_line);
    if (!rc)
        pr_warn("No chosen node found, continuing without\n");

    /* Initialize {size,address}-cells info */
    of_scan_flat_dt(early_init_dt_scan_root, NULL);
}
```

```

    /* Setup memory, calling early_init_dt_add_memory_arch */
    of_scan_flat_dt(early_init_dt_scan_memory, NULL);
}

```

3. **Device population:** After the board has been identified, and after the early configuration data has been parsed, then kernel initialization can proceed in the normal way. At some point in this process, `unflatten_device_tree()` is called to convert the data into a more efficient runtime representation. This is also when machine-specific setup hooks will get called, like `.init_early()`, `.init_irq()` and `.init_machine()` hooks on ARM devices. As can be guessed from the names, `.init_early()` is used for any machine-specific setup that needs to be executed early in the boot process, and `.init_irq()` is used to set up interrupt handling.

The most interesting hook in the DT context is `.init_machine()`, which is primarily responsible for populating the Linux device model with data about the platform. The list of devices can be obtained by parsing the DT and allocating device structures dynamically. The `of_platform_populate()` function, located in `drivers/of/platform.c`, walks through the nodes in the Device Tree and creates platform devices from it. The second argument to `of_platform_populate()` is an `of_device_id` table, and any node that matches an entry in that table will also get its child nodes registered.

A Device Tree can be considered as a language with its own syntax and semantics. The Device Tree data structure is a simple tree of nodes. The tree begins with a root node that doesn't have a name and it is marked with a slash (/) character. Each node may contain an arbitrary number of properties and next level nodes. Nodes and properties have names. Properties are named arrays of bytes, which may contain strings, numbers (big-endian), arbitrary sequences of bytes and any combination thereof. By analogy to a filesystem, nodes are directories and properties are files.

This is a syntactically correct Device Tree file that shows the nodes and property types:

```

/* SoC.dtsi */
/ {
    node1 {
        empty-property;
        cell-property = <100>;
        sub-node1 {
            cell-property = <1 2>;
            byte-property = [0x01 0x23 0x34 0x56];
            reference = <&primary>;
            list-property = "one", "two";
        };
        sub-node2 {
            string-property = "false";
        };
    };
    primary: node2 {
        mix-list-property = <4>, "Master", <5>;
    };
};

```

```

        node3 {
            string-list-property = "first string", "second string";
        };
};

```

The root node has 3 nodes, defined as node1, node2 and node3. The node2 has a label (primary). The node1 has two properties and two next level nodes (sub-node1 and sub-node2). In the node1, empty-property is handled as Boolean; if defined, it means true. In the node1, cell-property stores numeric values as 32-bit big-endian numbers. A reference in sub-node1 expresses an association with node2 through the primary label.

The property of node2 includes data of differing representations concatenated using a comma. The property of node3 includes a list of strings separated by commas.

This is another syntactically correct Device Tree file that includes the previous SoC.dtsi file:

```

/* board.dts */
/dts-v1/;
/include/ "SoC.dtsi"
/ {
    node1 {
        sub-node2 {
            string-property = "true";
        };
    };
    /delete-node/ node3;
};

```

The /dts-v1/ directive describes the used format of the Device Tree source. It's a required header and must be given only once.

The /include/ directive includes another file. Since every file defines a whole root node, every definition is handled like a transparency that adds more definitions and details to the Device Tree. Given that nodes are named, potentially with absolute paths, it is possible for the same node to appear twice in a DT file (and its inclusions). When this happens, the nodes and properties are combined, interleaving and overwriting properties as required (later values override earlier ones). The board.dts file redefines /node1/sub-node2/string-property value. The previously defined "false" will be replaced with "true" since the board.dts file will be processed after the SoC.dtsi file.

The /delete-node/ directive removes node3. A /delete-property/ directive could be used to delete a property.

The Device Tree Compiler (dtc) compiles the DT source into a binary form (.dtb). It exhibits design similarities to a C compiler: both can evaluate complicated numerical expressions, and both cannot process alphanumeric definitions and macros. The source code of the Device Tree Compiler is located in scripts/dtc/ in the kernel source tree. The Device Tree files are normally compiled in arch/arm/boot/dts/ in the kernel source tree. A Makefile file in that directory is prepared to invoke

a preprocessor and to search all needed include directories. It is easy to add new Device Tree files to the Makefile. The dtb is automatically built when the kernel image is built using the "make" command. However, to build just the dtb, the following command can be used from within the top Linux directory:

```
~/linux_rpi3/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
```

In the next chapters, you will learn about Device Tree special properties (for example, reg and interrupts) and how to read their values from the Device Tree by using different kernel functions (for example, platform_get_resource() and platform_get_irq()).

You can learn more about the usage of the Device Tree on Raspberry Pi at <https://www.raspberrypi.org/documentation/configuration/device-tree.md>.

3

The Simplest Drivers

A key concept in the design of an embedded Linux system is the separation of user applications from the underlying hardware. User space applications are not allowed to access peripheral registers, storage media or even RAM memory directly. Instead, the hardware is accessed via kernel drivers, and RAM memory is managed by the **memory management unit (MMU)**, with applications operating on **virtual addresses**.

This separation provides robustness. If it is assumed that the Linux kernel is operating correctly, then allowing only the kernel to interact with underlying hardware keeps applications from accidentally or maliciously misconfiguring hardware peripherals and placing them in unknown states.

This separation also provides portability. If only the kernel drivers manage the hardware specific code, only these drivers need to be modified in order to port a system from one hardware platform to another. Applications access a set of driver APIs that is consistent across hardware platforms, allowing applications to be moved from one platform to another with little or no modification to the source code.

Device drivers can be kernel modules or statically built into the kernel image. The default kernel builds most drivers into the kernel statically, so they are started automatically. A kernel module is not necessarily a device driver; it is an extension of the kernel. The kernel modules are loaded into virtual memory of the kernel. Building a device driver as a module makes the development easier, since it can be loaded, tested and unloaded without rebooting the kernel. The kernel modules are usually located in `/lib/modules/<kernel_version>/` on the root filesystem.

Every Linux kernel module has an `init()` and an `exit()` function. The `init()` function is called once when the driver is loaded, and the `exit()` function is called when the driver is removed. The `init()` function lets the OS know what the driver is capable of and which of its function must be called when a certain event takes place (for example, register a driver to the bus or register a char device). The `exit()` function must free all the resources that were requested by the `init()` function.

Macros `module_init()` and `module_exit()` export the symbols for the `init()` and `exit()` functions such that the kernel code that loads your module can identify these entry points.

There is a collection of macros used to identify various attributes of a module. These strings get packaged into the module and can be accessed by various tools. The most important module description macro is the `MODULE_LICENSE` macro. If this macro is not set to some sort of GPL license tag, then the kernel will become tainted when you load your module. When the kernel is tainted, it means that it is in a state that is not supported by the community. Most kernel developers will ignore bug reports involving tainted kernels, and community members may ask that you correct the tainting condition before they can proceed with diagnosing problems related to the kernel. In addition, some debugging functionality and API calls may be disabled when the kernel is tainted.

Licensing

The Linux kernel is licensed under the GNU General Public License version 2. This license gives you the right to use, study, modify and share the software freely. However, when the software is redistributed, modified or unmodified, the GPL requires that you redistribute the software under the same license, with the source code. If modifications are made to the Linux kernel (for example to adapt it to your hardware), it is a derivative work of the kernel and therefore must be released under GPLv2. However, you're only required to do so at the time the device starts to be distributed to your customers, not to the entire world.

The kernel modules provided in this book are released under the GPL license. For more information on open source software licenses, please see <http://opensource.org/licenses>.

LAB 3.1: "helloworld" module

In your first kernel module, you will simply send some info to the Raspberry Pi console every time you load and unload the module. The `hello_init()` and `hello_exit()` functions include a `pr_info()` function. This is much like the **printf** syntax you use in user applications, except that `pr_info()` is used to print log messages in kernel space. If you look into the real kernel code, you will always see something like:

```
printk(KERN_ERR "something went wrong, return code: %d\n",ret);
```

Where `KERN_ERR` is one of the eight different log levels defined in `include/linux/kern_levels.h` in the kernel source tree and specifies the severity of the error message. The **pr_*** macros (defined in `include/linux/printk.h` in the kernel source tree) are simple shorthand definitions for their respective **printk** calls and should be used in newer drivers.

In the home folder of your host PC, you will create the `linux_5.4_rpi3_drivers` folder, where you are going to store all the drivers and applications developed through this book.

```
~$ mkdir linux_5.4_rpi3_drivers
```


Create the `helloworld_rpi3.c` and Makefile files using your favorite text editor, and save them in the `linux_5.4_rpi3_drivers` folder. Write the Listing 3-1 code and the Listing 3-2 code to these files.

Secure Copy (SCP) will be added to the Makefile to transfer the modules from the host PC to the Raspberry Pi filesystem via Ethernet:

```
scp *.ko root@10.0.0.10:
```

The same Makefile will be reused for many of the labs by simply adding the new `<module name>.o` to the Makefile variable `obj-m`.

Compile the `helloworld_rpi3.c` driver, and deploy it to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

Listing 3-1: helloworld_rpi3.c

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

static int __init hello_init(void)
{
    pr_info("Hello world init\n");
    return 0;
}

static void __exit hello_exit(void)
{
    pr_info("Hello world exit\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");

MODULE_DESCRIPTION("This is a print out Hello World module");
```

Listing 3-2: Makefile

```
obj-m := helloworld.o

KERNEL_DIR ?= $(HOME)/linux_rpi3/linux

all:
    make -C $(KERNEL_DIR) \
        ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- \
        M=$(PWD) modules
```

```
clean:
    make -C $(KERNEL_DIR) \
        ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- \
        M=$(PWD) clean

deploy:
    scp *.ko root@10.0.0.10:/home/pi
```

helloworld_rpi3.ko demonstration

Load the module:

```
root@raspberrypi:/home/pi# insmod helloworld_rpi3.ko
Hello world init
```

See the MODULE macros defined in your module:

```
root@raspberrypi:/home/pi# modinfo helloworld_rpi3.ko
filename:      /home/pi/helloworld_rpi3.ko
description:   This is a print out Hello World module
author:        Alberto Liberal <aliberal@arroweurope.com>
license:       GPL
srcversion:    F329256EE5AE6A9F62F65DF
depends:
name:          helloworld_rpi3
vermagic:      5.4.83-v7+ SMP mod_unload modversions ARMv7 p2v8
```

An externally-built (“out-of-tree”) untainted module has been loaded:

```
root@raspberrypi:/home/pi# cat /sys/module/helloworld_rpi3/taint
0
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod helloworld_rpi3.ko
Hello world exit
```

Comment out the MODULE_LICENSE macro in the helloworld_rpi3.c file. Build, deploy and load the module again. Reboot the Pi!. Work with your tainted module (P0):

```
root@raspberrypi:/home/pi# reboot
```

```
root@raspberrypi:/home/pi# insmod helloworld_rpi3.ko
helloworld_rpi3: module license 'unspecified' taints kernel.
Disabling lock debugging due to kernel taint
Hello world init
```

```
root@raspberrypi:/home/pi# cat /proc/sys/kernel/tainted
P0
```

```
root@raspberrypi:/home/pi# cat /proc/modules
helloworld_rpi3 16384 0 - Live 0x7f1b6000 (P0)
```

Find your module in the sysfs:

```
root@raspberrypi:/home/pi# find /sys -name "*helloworld*"
/sys/module/helloworld_rpi3
```

```
root@raspberrypi:/home/pi# ls /sys/module/helloworld_rpi3/
coresize  initsize  notes     sections  taint
holders  initstate  refcnt    srcversion  uevent
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod helloworld_rpi3.ko
Hello world exit
```

LAB 3.2: "helloworld with parameters" module

Many **Linux loadable kernel modules (LKMs)** have parameters that can be set at load time, boot time and sometimes at run-time. In this kernel module, you are going to pass a parameter in the command line that will be set during the module loading. You can also read the parameters via the sysfs filesystem.

As you saw in the section "The sysfs filesystem" of Chapter 2, the sysfs is a virtual filesystem provided by the Linux kernel that exports information about the various kernel subsystems, hardware devices and associated device drivers from the kernel's device model to the user space through virtual files. In addition to providing information about various devices and kernel subsystems, exported virtual files are also used for their configuration.

The definition of module parameters is done via the macro `module_param()`:

```
/*
 * the perm argument specifies the permissions
 * of the corresponding file in sysfs.
 */
module_param(name, type, perm);
```

The main code sections of the driver will now be described:

1. After the `#include` statements, declare a new `num` variable, and use the `module_param()` on it:

```
static int num = 5;

/* S_IRUGO: everyone can read the sysfs entry */
module_param(num, int, S_IRUGO);
```

2. Write the `pr_info` statement in the `hello_init()` function, as shown below:

```
pr_info("parameter num = %d.\n", num);
```

3. Create a new `helloworld_rpi3_with_parameters.c` file in the `linux_5.4_rpi3_drivers` folder, and add `helloworld_rpi3_with_parameters.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
```

```
~/linux_5.4_rpi3_drivers$ make deploy
```

Listing 3-3: helloworld_rpi3_with_parameters.c

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

static int num = 5;

module_param(num, int, S_IRUGO);

static int __init hello_init(void)
{
    pr_info("parameter num = %d\n", num);
    return 0;
}

static void __exit hello_exit(void)
{
    pr_info("Hello world with parameter exit\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a module that accepts parameters");
```

Listing 3-4: Makefile

```
obj-m := helloworld.o helloworld_rpi3_with_parameters.o

KERNEL_DIR ?= $(HOME)/linux_rpi3/linux

all:
    make -C $(KERNEL_DIR) \
        ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- \
        M=$(PWD) modules

clean:
    make -C $(KERNEL_DIR) \
        ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- \
        M=$(PWD) clean

deploy:
    scp *.ko root@10.0.0.10:/home/pi
```

helloworld_rpi3_with_parameters.ko demonstration

Load the module:

```
root@raspberrypi:/home/pi# insmod helloworld_rpi3_with_parameters.ko
parameter num = 5
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod helloworld_rpi3_with_parameters.ko
Hello world with parameter exit
```

Load the module with a parameter:

```
root@raspberrypi:/home/pi# insmod helloworld_rpi3_with_parameters.ko num=10
parameter num = 10
```

```
root@raspberrypi:/home/pi# cat /sys/module/helloworld_rpi3_with_parameters/parameters/num
10
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod helloworld_rpi3_with_parameters.ko
Hello world with parameter exit
```


4

Character Drivers

Typically, an operating system is designed to hide the underlying hardware details from the user application. Applications do, however, require the ability to access data that is captured by hardware peripherals, as well as the ability to drive peripherals with output. Since the peripheral registers are accessible only by the Linux kernel, only the kernel is able to collect data streams as they are captured by these peripherals.

Linux requires a mechanism to transfer data from kernel to user space. This transfer of data is handled via **device nodes**, which are also known as **virtual files**. Device nodes exist within the root filesystem, though they are not true files. When a user reads from a device node, the kernel copies the data stream captured by the underlying driver into the application memory space. When a user writes to a device node, the kernel copies the data stream provided by the application into the data buffers of the driver, which are eventually output via the underlying hardware. These virtual files can be "opened" and "read from" or "written to" by the user application using standard **system calls**.

Each device has a unique driver that handles requests from user applications that are eventually passed to the core. Linux supports three types of devices: **character devices**, **block devices** and **network devices**. While the concept is the same, the difference in the drivers for each of these devices is the manner in which the files are "opened" and "read from" or "written to". Character devices are the most common devices, which are read and written directly without buffering, for example, keyboards, monitors, printers and serial ports. Block devices can only be written to and read from in multiples of the block size, typically 512 or 1024 bytes. They may be randomly accessed, i.e., any block can be read or written no matter where it is on the device. A classic example of a block device is a hard disk drive. Network devices are accessed via the BSD socket interface and the networking subsystems.

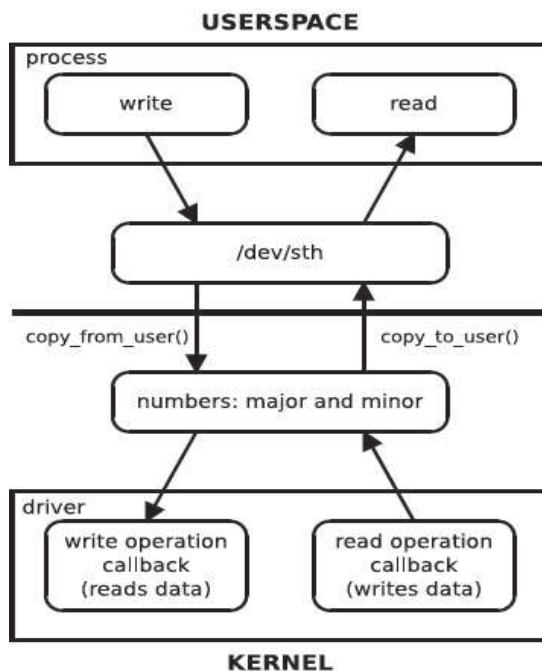
Character devices are identified by a **c** in the first column of a listing, and block devices are identified by a **b**. The access permissions, owner and group of the device are provided for each device.

From the point of view of an application, a character device is essentially a file. A process only knows a /dev file path. The process opens the file by using the `open()` system call and performs standard file operations like `read()` and `write()`.

In order to achieve this, a character driver must implement the operations described in the `file_operations` structure (declared in `include/linux/fs.h` in the kernel source tree) and register them. In the `file_operations` structure shown below, you can see some of the most common operations for a character driver:

```
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 *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
};
```

The Linux filesystem layer will ensure that these operations are called when a user space application makes the corresponding system call (on the kernel side, the driver implements and registers the callback operation).



The kernel driver will use the specific functions `copy_from_user()` and `copy_to_user()` to exchange data with user space, as shown in the previous figure.

Both the `read()` and `write()` methods return a negative value if an error occurs. A return value greater than or equal to 0, instead, tells the calling program how many bytes have been successfully transferred. If some data is transferred correctly and then an error happens, the return value must be the count of bytes successfully transferred, and the error does not get reported until the next time the function is called. Implementing this convention requires, of course, that your driver remember that the error has occurred so that it can return the error status in the future.

The return value for `read()` is interpreted by the calling application program:

1. If the value equals the count argument passed to the `read` system call, the requested number of bytes has been transferred. This is the optimal case.
2. If the value is positive, but smaller than the count, then only part of the data has been transferred. This may happen for a number of reasons, depending on the device. Most often, the application program retries the read. For instance, if you read using the `fread()` function, the library function reissues the system call until completion of the requested data transfer. If the value is 0, end-of-file was reached.
3. A negative value means there was an error. The value specifies what the error was, according to `<linux/errno.h>`. Typical values returned on error include `-EINTR` (interrupted system call) or `-EFAULT` (bad address).

In Linux, every device is identified by two numbers: a **major** number and a **minor** number. These numbers can be seen by invoking `ls -l /dev` on the host PC. Every device driver registers its major number with the kernel and is completely responsible for managing its minor numbers. When accessing a device file, the major number selects which device driver is being called to perform the input/output operation. The major number is used by the kernel to identify the correct device driver when the device is accessed. The role of the minor number is device dependent and is handled internally within the driver. For instance, the Raspberry Pi has several hardware UART ports. The same driver can be used to control all the UARTs, but each physical UART needs its own device node, so the device nodes for these UARTs will all have the same major number, but will have unique minor numbers.

LAB 4.1: "helloworld character" module

Linux systems in general traditionally used a static device creation method, whereby a great number of device nodes were created under `/dev` (sometimes literally thousands of nodes), regardless of whether or not the corresponding hardware devices actually existed. This was typically done via a `MAKEDEV` script, which contains a number of calls to the `mknod` program

with the relevant major and minor device numbers for every possible device that might exist in the world.

This is not the right approach to create device nodes nowadays, as you have to create a block or character device file entry manually and associate it with the device, as shown in the Raspberry Pi's terminal command line below:

```
root@raspberrypi:/home/pi# mknod /dev/mydev c 202 108
```

Despite all this, you will develop your next driver using this static method, purely for educational purposes, and you will see in the few next drivers a better way to create the device nodes by using **devtmpfs** and the **miscellaneous framework**.

In this kernel module lab, you will interact with user space through an `ioctl_test` user application. You will use `open()` and `ioctl()` system calls in your application, and you will write its corresponding driver's callback operations on the kernel side, providing a communication between user and kernel space.

In the first lab, you saw what a basic driver looks like. This driver didn't do much except printing some text during installation and removal. In the following lab, you will expand this driver to create a device with a major and minor number. You will also create a user application to interact with the driver. Finally, you will handle file operations in the driver to service requests from user space.

In the kernel, a character-type device is represented by `struct cdev`, a structure used to register it in the system.

Registration and unregistration of character devices

The registration/unregistration of a character device is made by specifying the major and minor. The `dev_t` type is used to keep the identifiers of a device (both major and minor) and can be obtained by using the `MKDEV` macro.

For the static assignment and unallocation of device identifiers, the `register_chrdev_region()` and `unregister_chrdev_region()` functions are used. The first device identifier is obtained by using the `MKDEV` macro.

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

It is recommended that device identifiers be dynamically assigned by using the `alloc_chrdev_region()` function. This function allocates a range of char device numbers. The major number will be chosen dynamically and returned (along with the first minor number) in `dev`. This function returns zero or a negative error code.

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

See below the description of the arguments of the previous function:

- `dev`: Output parameter for first assigned number.
- `baseminor`: First of the requested range of minor numbers.
- `count`: Number of minor numbers required.
- `name`: Name of the associated device or driver.

In the line of code below, the second parameter reserves a number (`my_minor_count`) of devices, starting with `my_major` major and `my_first_minor` minor. The first parameter of the `register_chrdev_region()` function is the first identifier of the device. The successive identifiers can be retrieved by using the `MKDEV` macro.

```
register_chrdev_region(MKDEV(my_major, my_first_minor), my_minor_count, "my_device_driver");
```

After assigning the identifiers, the character device will have to be initialized by using the `cdev_init()` function and registered to the kernel by using the `cdev_add()` function. The `cdev_init()` and `cdev_add()` functions will be called as many times as assigned device identifiers.

The following sequence registers and initializes a number (`MY_MAX_MINORS`) of devices:

```
#include <linux/fs.h>
#include <linux/cdev.h>

#define MY_MAJOR      42
#define MY_MAX_MINORS 5

struct my_device_data {
    struct cdev cdev;
    /* my data starts here */
    [...]
};

struct my_device_data devs[MY_MAX_MINORS];

const struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .read = my_read,
    .write = my_write,
    .release = my_release,
    .unlocked_ioctl = my_ioctl
};
```

```

int init_module(void)
{
    int i, err;
    register_chrdev_region(MKDEV(MY_MAJOR, 0), MY_MAX_MINORS, "my_device_driver");

    for(i = 0; i < MY_MAX_MINORS; i++) {
        /* initialize devs[i] fields and register character devices */
        cdev_init(&devs[i].cdev, &my_fops);
        cdev_add(&devs[i].cdev, MKDEV(MY_MAJOR, i), 1);
    }

    return 0;
}

```

The following code snippet deletes and unregisters the character devices:

```

void cleanup_module(void)
{
    int i;

    for(i = 0; i < MY_MAX_MINORS; i++) {
        /* release devs[i] fields */
        cdev_del(&devs[i].cdev);
    }
    unregister_chrdev_region(MKDEV(MY_MAJOR, 0), MY_MAX_MINORS);
}

```

The main code sections of the driver will now be described:

1. Include the following header files to support character devices:

```

#include <linux/cdev.h>
#include <linux/fs.h>

```

2. Define the major number:

```

#define MY_MAJOR_NUM 202

```

3. One of the first things your driver will need to do when setting up a char device is to obtain one or more device identifiers (major and minor numbers) to work with. The necessary function for this task is `register_chrdev_region()`, which is declared in `include/linux/fs.h` in the kernel source tree. Add the following lines of code to the `hello_init()` function to allocate the device numbers when the module is loaded. The `MKDEV` macro will combine a major number and a minor number to a `dev_t` data type that is used to hold the first device identifier.

```
dev_t dev = MKDEV(MY_MAJOR_NUM, 0); /* get first device identifier */
/*
 * Allocates all the character device identifiers,
 * only one in this case, the one obtained with the MKDEV macro
 */
register_chrdev_region(dev, 1, "my_char_device");
```

4. Add the following line of code to the `hello_exit()` function to return the devices when the module is removed:

```
unregister_chrdev_region(MKDEV(MY_MAJOR_NUM, 0), 1);
```

5. Create a `file_operations` structure called `my_dev_fops`. This structure defines function pointers for "opening", "reading from", "writing to" the device, etc.

```
static const struct file_operations my_dev_fops = {
    .owner = THIS_MODULE,
    .open = my_dev_open,
    .release = my_dev_close,
    .unlocked_ioctl = my_dev_ioctl,
};
```

6. Implement each of the callback functions that are declared in the `file_operations` structure:

```
static int my_dev_open(struct inode *inode, struct file *file)
{
    pr_info("my_dev_open() is called.\n");
    return 0;
}

static int my_dev_close(struct inode *inode, struct file *file)
{
    pr_info("my_dev_close() is called.\n");
    return 0;
}

static long my_dev_ioctl(struct file *file, unsigned int cmd,
unsigned long arg)
{
    pr_info("my_dev_ioctl() is called. cmd = %d, arg = %ld\n", cmd, arg);
    return 0;
}
```

7. Add these file operation functionalities to your character device. The Linux kernel uses `struct cdev` to represent character devices internally; therefore you will create a `struct cdev` variable called `my_dev` and initialize it by using the `cdev_init()` function call, which takes the variable `my_dev` and the structure `my_dev_fops` as parameters. Once the `cdev` structure is setup, you will tell the kernel about it by using the `cdev_add()` function. You will call `cdev_init()` and `cdev_add()` as many times as allocated character device identifiers (only once in this driver).

```
static struct cdev my_dev;
cdev_init(&my_dev, &my_dev_fops);
ret= cdev_add(&my_dev, dev, 1);
```

8. Add the following line of code to the `hello_exit()` function to delete the `cdev` structure:

```
cdev_del(&my_dev);
```

9. Once the kernel module has been dynamically loaded, the user needs to create a device node to reference the driver. Linux provides the `mknod` utility for this purpose. The `mknod` command has four parameters. The first parameter is the name of the device node that will be created. The second parameter indicates whether the driver to which the device node interfaces is a block driver or character driver. The last two parameters passed to `mknod` are the major and minor numbers. Assigned major numbers are listed in the `/proc/devices` file and can be viewed using the `cat` command. The created device node should be placed in the `/dev` directory.
10. Create a new `helloworld_rpi3_char_driver.c` file in the `linux_5.4_rpi3_drivers` folder, and write the Listing 4-1 code on it. Add `helloworld_rpi3_char_driver.o` to your Makefile `obj-m` variable.
11. In the `linux_5.4_rpi3_drivers` folder, you will create the `apps` folder, where you are going to store most of the applications developed through this book.

```
~/linux_5.4_rpi3_drivers$ mkdir apps
```

12. In the `apps` folder, you will create an `ioctl_test.c` file, then write the Listing 4-3 code on it. You will also create a Makefile (Listing 4-2) file in the `apps` folder to compile and deploy the application.
13. Compile the `helloworld_rpi3_char_driver.c` driver and the `ioctl_test.c` application, and deploy them to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
~/linux_5.4_rpi3_drivers/apps$ make
~/linux_5.4_rpi3_drivers/apps$ make deploy
```

Listing 4-1: `helloworld_rpi3_char_driver.c`

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

/* Add header files to support character devices */
#include <linux/cdev.h>
#include <linux/fs.h>

/* Define mayor number */
#define MY_MAJOR_NUM 202
```

```

static struct cdev my_dev;

static int my_dev_open(struct inode *inode, struct file *file)
{
    pr_info("my_dev_open() is called.\n");
    return 0;
}

static int my_dev_close(struct inode *inode, struct file *file)
{
    pr_info("my_dev_close() is called.\n");
    return 0;
}

static long my_dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    pr_info("my_dev_ioctl() is called. cmd = %d, arg = %ld\n", cmd, arg);
    return 0;
}

/* Declare a file_operations structure */
static const struct file_operations my_dev_fops = {
    .owner = THIS_MODULE,
    .open = my_dev_open,
    .release = my_dev_close,
    .unlocked_ioctl = my_dev_ioctl,
};

static int __init hello_init(void)
{
    int ret;

    /* Get first device identifier */
    dev_t dev = MKDEV(MY_MAJOR_NUM, 0);
    pr_info("Hello world init\n");

    /* Allocate a number of devices */
    ret = register_chrdev_region(dev, 1, "my_char_device");
    if (ret < 0) {
        pr_info("Unable to allocate mayor number %d\n", MY_MAJOR_NUM);
        return ret;
    }

    /* Initialize the cdev structure and add it to kernel space */
    cdev_init(&my_dev, &my_dev_fops);
    ret = cdev_add(&my_dev, dev, 1);
    if (ret < 0) {
        unregister_chrdev_region(dev, 1);
        pr_info("Unable to add cdev\n");
        return ret;
    }

    return 0;
}

```

```
static void __exit hello_exit(void)
{
    pr_info("Hello world exit\n");
    cdev_del(&my_dev);
    unregister_chrdev_region(MKDEV(MY_MAJOR_NUM, 0), 1);
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a module that interacts with the ioctl system call");
```

Listing 4-2: Makefile

```
CC = arm-linux-gnueabi-hf-gcc

all: ioctl_test

app : ioctl_test.c
    $(CC) -o $@ $^
clean :
    rm ioctl_test
deploy : ioctl_test
    scp $^ root@10.0.0.10:/home/pi
```

Listing 4-3: ioctl_test.c

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    /* First, you need run "mknod /dev/mydev c 202 0" to create /dev/mydev */

    int my_dev = open("/dev/mydev", 0);

    if (my_dev < 0) {
        perror("Fail to open device file: /dev/mydev.");
    } else {
        ioctl(my_dev, 100, 110); /* cmd = 100, arg = 110. */
        close(my_dev);
    }

    return 0;
}
```


helloworld_rpi3_char_driver.ko demonstration

Load the module:

```
root@raspberrypi:/home/pi# insmod helloworld_rpi3_char_driver.ko
Hello world init
```

See the allocated "my_char_device". The device mydev is not created under /dev yet:

```
root@raspberrypi:/home/pi# cat /proc/devices
Character devices:
 1 mem
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 [...]
202 my_char_device
 [...]
```

```
root@raspberrypi:/home/pi# ls -l /dev/
```

Create mydev under /dev using mknod, and verify its creation:

```
root@raspberrypi:/home/pi# mknod /dev/mydev c 202 0

root@raspberrypi:/home/pi# ls -l /dev/mydev
crw-r--r-- 1 root root 202, 0 Apr  8 19:00 /dev/mydev
```

Run the application ioctl_test:

```
root@raspberrypi:/home/pi# ./ioctl_test
my_dev_open() is called.
my_dev_ioctl() is called. cmd = 100, arg = 110
my_dev_close() is called.
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod helloworld_rpi3_char_driver.ko
Hello world exit
```

Creating device files with devtmpfs

Before Linux 2.6.32, on basic Linux systems, the device files had to be created manually by using the mknod command. The coherency between device files and devices handled by the kernel was left to the system developer. With the release of the 2.6 series of stable kernel, a new ram-based filesystem called **sysfs** came about. The job of sysfs is to export a view of the system's hardware configuration to the user space processes.

Linux drivers compiled into the kernel register their objects with a sysfs as they are detected by the kernel. For Linux drivers compiled as modules, this registration will happen when the module is loaded. Sysfs is enabled and ready to be used via the Linux kernel configuration CONFIG_SYSFS, which should be set to yes by default.

Device files are created by the kernel via the temporary **devtmpfs** filesystem. Any driver that wishes to register a device node will use devtmpfs (via the core driver) to do it. When a devtmpfs instance is mounted on `/dev`, the device node will initially be created with a fixed name, permissions and owner. These entries can be both read from and written to. All device nodes are owned by root and have the default mode of 0600.

Shortly afterward, the kernel will send an uevent to udevd. Based on the rules specified in the files within the `/etc/udev/rules.d/`, `/lib/udev/rules.d/` and `/run/udev/rules.d/` directories, udevd will create additional symlinks to the device node, change its permissions, owner or group, or modify the internal udevd database entry (name) for that object. The rules in these three directories are numbered, and all three directories are merged together. If udevd can't find a rule for the device it is creating, it will leave the permissions and ownership at whatever devtmpfs used initially.

The `CONFIG_DEVTMPFS_MOUNT` kernel configuration option makes the kernel mounts devtmpfs automatically at boot time, except when booting on an initramfs.

LAB 4.2: "class character" module

In this kernel module lab, you will use your previous `helloworld_rpi3_char_driver` driver as a starting point, but this time, the device node will be created by using **devtmpfs** instead of doing it manually.

You will add an entry in the `/sys/class/` directory. The `/sys/class/` directory offers a view of the device drivers grouped by classes.

When the `register_chrdev_region()` function tells the kernel that there is a driver with a specific major number, it doesn't specify anything about the type of driver, so it will not create an entry under `/sys/class/`. This entry is necessary so that devtmpfs can create a device node under `/dev`. Drivers will have a class name and a device name under the `/sys` folder for each created device.

A Linux driver creates/destroys the class by using the following kernel APIs:

```
class_create() /* creates a class for your devices visible in /sys/class/ */
class_destroy() /* removes the class */
```

A Linux driver creates the device nodes by using the following kernel APIs:

```
device_create() /* creates a device node in the /dev directory */
device_destroy() /* removes a device node in the /dev directory */
```

The main points that differ from your previous `helloworld_rpi3_char_driver` driver will now be described:

1. Include the following header file to create the class and device files:

```
#include <linux/device.h> /* class_create(), device_create() */
```

2. Your driver will have a class name and a device name; `hello_class` is used as the class name and `mydev` as the device name. This results in the creation of a device that appears on the file system at `/sys/class/hello_class/mydev`. Add the following definitions for the device and class names:

```
#define DEVICE_NAME "mydev"
#define CLASS_NAME "hello_class"
```

3. The `hello_init()` function is longer than the one written in the `helloworld_rpi3_char_driver` driver. That is because it now automatically allocates a major number to the device by using the function `alloc_chrdev_region()`, as well as registering the device class and creating the device node.

```
static int __init hello_init(void)
{
    dev_t dev_no;
    int Major;
    struct device* helloDevice;

    /* Allocate dynamically device numbers (only one in this driver) */
    ret = alloc_chrdev_region(&dev_no, 0, 1, DEVICE_NAME);

    /*
     * Get the device identifiers using MKDEV. We are doing this
     * for teaching purposes, as we only use one identifier in this
     * driver and dev_no could be used as a parameter for cdev_add()
     * and device_create() without needing to use the MKDEV macro
     */

    /* Get the mayor number from the first device identifier */
    Major = MAJOR(dev_no);

    /* Get the first device identifier, that matches with dev_no */
    dev = MKDEV(Major,0);

    /* Initialize the cdev structure, and add it to kernel space */
    cdev_init(&my_dev, &my_dev_fops);
    ret = cdev_add(&my_dev, dev, 1);

    /* Register the device class */
    helloClass = class_create(THIS_MODULE, CLASS_NAME);

    /* Create a device node named DEVICE_NAME associated to dev */
    helloDevice = device_create(helloClass, NULL, dev, NULL, DEVICE_NAME);

    return 0;
}
```

4. Create a new `helloworld_rpi3_class_driver.c` file in the `linux_5.4_rpi3_drivers` folder, and add `helloworld_rpi3_class_driver.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

Listing 4-4: helloworld_rpi3_class_driver.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>

#define DEVICE_NAME "mydev"
#define CLASS_NAME "hello_class"

static struct class* helloClass;
static struct cdev my_dev;
dev_t dev;

static int my_dev_open(struct inode *inode, struct file *file)
{
    pr_info("my_dev_open() is called.\n");
    return 0;
}

static int my_dev_close(struct inode *inode, struct file *file)
{
    pr_info("my_dev_close() is called.\n");
    return 0;
}

static long my_dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    pr_info("my_dev_ioctl() is called. cmd = %d, arg = %ld\n", cmd, arg);
    return 0;
}

/* Declare a file_operations structure */
static const struct file_operations my_dev_fops = {
    .owner          = THIS_MODULE,
    .open           = my_dev_open,
    .release        = my_dev_close,
    .unlocked_ioctl = my_dev_ioctl,
};

static int __init hello_init(void)
{
    int ret;
    dev_t dev_no;
    int Major;
    struct device* helloDevice;

    pr_info("Hello world init\n");
```

```

/* Allocate device numbers dynamically */
ret = alloc_chrdev_region(&dev_no, 0, 1, DEVICE_NAME);
if (ret < 0) {
    pr_info("Unable to allocate Mayor number \n");
    return ret;
}

/* Get the device identifiers */
Major = MAJOR(dev_no);
dev = MKDEV(Major,0);

pr_info("Allocated correctly with major number %d\n", Major);

/* Initialize the cdev structure and add it to kernel space */
cdev_init(&my_dev, &my_dev_fops);
ret = cdev_add(&my_dev, dev, 1);
if (ret < 0) {
    unregister_chrdev_region(dev, 1);
    pr_info("Unable to add cdev\n");
    return ret;
}

/* Register the device class */
helloClass = class_create(THIS_MODULE, CLASS_NAME);
if (IS_ERR(helloClass)) {
    unregister_chrdev_region(dev, 1);
    cdev_del(&my_dev);
    pr_info("Failed to register device class\n");
    return PTR_ERR(helloClass);
}
pr_info("device class registered correctly\n");

/* Create a device node named DEVICE_NAME associated to dev */
helloDevice = device_create(helloClass, NULL, dev, NULL, DEVICE_NAME);
if (IS_ERR(helloDevice)) {
    class_destroy(helloClass);
    cdev_del(&my_dev);
    unregister_chrdev_region(dev, 1);
    pr_info("Failed to create the device\n");
    return PTR_ERR(helloDevice);
}
pr_info("The device is created correctly\n");

return 0;
}

static void __exit hello_exit(void)
{
    device_destroy(helloClass, dev); /* remove the device */
    class_destroy(helloClass);      /* remove the device class */
    cdev_del(&my_dev);
    unregister_chrdev_region(dev, 1); /* unregister the device numbers */
    pr_info("Hello world with parameter exit\n");
}

```

```

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a module that interacts with the ioctl system call");

```

helloworld_rpi3_class_driver.ko demonstration

Reboot your Raspberry Pi to remove the mydev device that you created manually in the previous lab:

```
root@raspberrypi:/home/pi# reboot
```

Load the module:

```

root@raspberrypi:/home/pi# insmod helloworld_rpi3_class_driver.ko
Hello world init
Allocated correctly with major number 238
device class registered correctly
The device is created correctly

```

Check that hello_class and mydev are created, and see the entries under mydev:

```

root@raspberrypi:/home/pi# ls /sys/class
bcm2708_vcio    hello_class    iscsi_session  raw            thermal
bcm2835-gpiomem hidraw         iscsi_transport rc             tty
bdi            hwmon          leds           regulator      uio
block          i2c-adapter   lirc           rfkill         vc
bluetooth      i2c-dev       mdio_bus       rtc            vchiq
bsg            ieee80211     mem            scsi_device    vc-mem
devcoredump    input         misc           scsi_disk      vc-sm
dma            iscsi_connection mmc_host       scsi_host      video4linux
dma_heap       iscsi_endpoint net            sound          vtconsole
gpio           iscsi_host    power_supply   spi_master     watchdog
graphics       iscsi_iface   pwm            spi_slave

```

```

root@raspberrypi:/home/pi# ls -l /dev/mydev
crw----- 1 root root 238, 0 Apr  8 18:56 /dev/mydev

```

```

root@raspberrypi:/home/pi# ls /sys/class/hello_class/mydev
dev power subsystem uevent

```

See the assigned mayor and minor numbers for the device mydev:

```

root@raspberrypi:/home/pi# cat /sys/class/hello_class/mydev/dev
238:0

```

Run the application ioctl_test:

```

root@raspberrypi:/home/pi# ./ioctl_test
my_dev_open() is called.
my_dev_ioctl() is called. cmd = 100, arg = 110
my_dev_close() is called.

```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod helloworld_rpi3_class_driver.ko
Hello world with parameter exit
```

Miscellaneous character driver

The **Misc Framework** is an interface exported by the Linux kernel that allows modules to register their individual minor numbers.

The device driver implemented as a miscellaneous character uses the major number allocated by the Linux kernel for **miscellaneous devices**. This eliminates the need to define a unique major number for the driver; this is important, as a conflict between major numbers has become increasingly likely, and use of the misc device class is an effective tactic. Each probed device is dynamically assigned a minor number and is listed with a directory entry within the sysfs pseudo-filesystem under `/sys/class/misc/`.

Major number 10 is officially assigned to the misc driver. Modules can register individual minor numbers with the misc driver and take care of a small device, needing only a single entry point.

Registering a minor number

A `miscdevice` structure is declared in `include/linux/miscdevice.h` in the kernel source tree:

```
struct miscdevice {
    int minor;
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const char *nodename;
    umode_t mode;
};
```

Where:

- `minor` is the minor number being registered.
- `name` is the name for this device, found in the `/proc/misc` file.
- `fops` is a pointer to the `file_operations` structure.
- `parent` is a pointer to a device structure that represents the hardware device exposed by this driver.

The misc driver exports two functions, `misc_register()` and `misc_deregister()`, to register and unregister their own minor number. These functions are declared in `include/linux/miscdevice.h` and defined in `drivers/char/misc.c` in the kernel source tree:

```
int misc_register(struct miscdevice *misc);
int misc_deregister(struct miscdevice *misc);
```

The `misc_register()` function registers a miscellaneous device with the kernel. If the minor number is set to `MISC_DYNAMIC_MINOR`, a minor number is dynamically assigned and placed in the minor field of the `miscdevice` structure. In other cases, the minor number requested is used.

The structure passed as an argument is linked into the kernel and may not be destroyed until it has been unregistered. By default, an `open()` syscall to the device sets the `file->private_data` to point to the structure. A zero is returned on success and a negative `errno` code for failure.

The typical code snippet for assigning a dynamic minor number is as follows:

```
static struct miscdevice my_dev;

int init_module(void)
{
    my_dev.minor = MISC_DYNAMIC_MINOR;
    my_dev.name = "my_device";
    my_dev.fops = &my_fops;
    misc_register(&my_dev);
    pr_info("my: got minor %i\n", my_dev.minor);
    return 0;
}
```

LAB 4.3: "miscellaneous character" module

In this lab, you will use your previous `helloworld_rpi3_char_driver` driver as a starting point. You will achieve the same result through the `misc` framework, but you will write fewer lines of code!!

The main code sections of the driver will now be described:

1. Add the header file that declares the `miscdevice` structure:

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

2. Initialize the `miscdevice` structure:

```
static struct miscdevice helloworld_miscdevice = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "mydev",
    .fops = &my_dev_fops,
}
```

3. Register and unregister the device with the kernel:

```
misc_register(&helloworld_miscdevice);
misc_deregister(&helloworld_miscdevice);
```

4. Create a new `misc_rpi3_driver.c` file in the `linux_5.4_rpi3_drivers` folder, and add `misc_rpi3_driver.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
```



```
~/linux_5.4_rpi3_drivers$ make deploy
```

Listing 4-5: misc_rpi3_driver.c

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

static int my_dev_open(struct inode *inode, struct file *file)
{
    pr_info("my_dev_open() is called.\n");
    return 0;
}

static int my_dev_close(struct inode *inode, struct file *file)
{
    pr_info("my_dev_close() is called.\n");
    return 0;
}

static long my_dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    pr_info("my_dev_ioctl() is called. cmd = %d, arg = %ld\n", cmd, arg);
    return 0;
}

static const struct file_operations my_dev_fops = {
    .owner = THIS_MODULE,
    .open = my_dev_open,
    .release = my_dev_close,
    .unlocked_ioctl = my_dev_ioctl,
};

/* Declare and initialize struct miscdevice */
static struct miscdevice helloworld_miscdevice = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "mydev",
    .fops = &my_dev_fops,
};

static int __init hello_init(void)
{
    int ret_val;
    pr_info("Hello world init\n");

    /* Register the device with the kernel */
    ret_val = misc_register(&helloworld_miscdevice);

    if (ret_val != 0) {
        pr_err("could not register the misc device mydev");
        return ret_val;
    }
}
```

```

    pr_info("mydev: got minor %i\n", helloworld_miscdevice.minor);
    return 0;
}

static void __exit hello_exit(void)
{
    pr_info("Hello world exit\n");

    /* Unregister the device with the Kernel */
    misc_deregister(&helloworld_miscdevice);
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is the helloworld_char_driver using misc framework");

```

misc_rpi3_driver.ko demonstration

Load the module:

```

root@raspberrypi:/home/pi# insmod misc_rpi3_driver.ko
Hello world init
mydev: got minor 60

```

Check that mydev is created under the misc class folder and under the /dev folder:

```

root@raspberrypi:/home/pi# ls /sys/class/misc
autofs      cpu_dma_latency  hw_random      mydev      vscm-cma
cachefiles  fuse             loop-control   rfkill     watchdog
root@raspberrypi:/home/pi# ls -l /dev/mydev
crw----- 1 root root 10, 60 Apr  8 18:59 /dev/mydev

```

Run the application ioctl_test:

```

root@raspberrypi:/home/pi# ./ioctl_test
my_dev_open() is called.
my_dev_ioctl() is called. cmd = 100, arg = 110
my_dev_close() is called.

```

Remove the module:

```

root@raspberrypi:/home/pi# rmmod misc_rpi3_driver.ko
Hello world exit

```

5

Platform Drivers

So far, you have been building your driver as a loadable driver module which was loaded during run time. The character driver is complete and has been tested thoroughly with a user space application. In your next assignment, you will convert the character driver to a **platform driver**. On embedded systems, devices are often not connected through a bus, allowing enumeration or hotplugging for these devices.

However, you still want all of these devices to be part of the device model. Such devices, instead of being dynamically detected, must be statically described by using two different methods:

1. By **direct instantiation** of `platform_device` structures, as done on a few old ARM non-Device Tree based platforms. The definition is done in the board-specific or SoC specific code.
2. In the **Device Tree**, a hardware description file used on some architectures. The device driver matches with the physical devices described in the `.dts` file. After this matching, the driver's `probe()` function is called. An `.of_match_table` has to be included in the driver code to allow this matching.

Amongst the non-discoverable devices, a huge family is directly part of a system-on-chip: UART controllers, Ethernet controllers, SPI controllers, graphic or audio devices, etc. In the Linux kernel, a special bus, called the **platform bus**, has been created to handle such devices. It supports platform drivers that handle platform devices. It works like any other bus (USB, PCI), except that the devices are enumerated statically instead of being discovered dynamically.

Each platform driver is responsible for instantiating and registering an instance of a `platform_driver` structure within the device model core. Platform drivers follow the standard driver model convention, where discovery/enumeration is handled outside the drivers, and drivers provide `probe()` and `remove()` methods. Platform drivers support power management and shutdown notifications using the standard conventions. The most important members of the `platform_driver` structure are shown below:

```
struct platform_driver {  
    int (*probe)(struct platform_device *);  
};
```

```

int (*remove)(struct platform_device *);
void (*shutdown)(struct platform_device *);
int (*suspend)(struct platform_device *, pm_message_t state);
int (*suspend_late)(struct platform_device *, pm_message_t state);
int (*resume_early)(struct platform_device *);
int (*resume)(struct platform_device *);
struct device_driver driver;
};

```

At a minimum, the `probe()` and `remove()` callbacks must be supplied. The `probe()` function is called when the "bus driver" pairs the "device" to the "device driver". The `probe()` function is responsible for initializing the device and registering it in the appropriate kernel framework:

1. It gets a pointer to a device structure as an argument (for example, `struct pci_dev *`, `struct usb_dev *`, `struct platform_device *`, `struct i2c_client *`).
2. It initializes the device, maps I/O memory, allocates buffers, registers interrupt handlers, etc.
3. It registers the device to specific framework(s).

The `suspend()` and `resume()` functions are used by devices that support low power management features.

The platform driver responsible for the platform device should be registered to the platform core by using the `platform_driver_register()` function. Register your platform driver in the module `init()` function, and unregister it in the module `exit()` function, as shown in the following code snippet:

```

static int hello_init(void)
{
    pr_info("demo_init enter\n");
    platform_driver_register(&my_platform_driver);
    pr_info("hello_init exit\n");
    return 0;
}

static void hello_exit(void)
{
    pr_info("demo_exit enter\n");
    platform_driver_unregister(&my_platform_driver);
    pr_info("demo_exit exit\n");
}

module_init(hello_init);
module_exit(hello_exit);

```

You can also use the `module_platform_driver()` macro. This is a helper macro for drivers that don't do anything special in the 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()`.

```

/*
 * 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)

```

LAB 5.1: "platform device" module

The functionality of this platform driver is the same as the misc char driver, but this time you will register your char device in the `probe()` function instead of the `init()` function. When the kernel module is loaded, the **platform device driver** registers itself with the **platform bus driver** by using the `platform_driver_register()` function. The `probe()` function is called when the platform device driver matches the value of one of its compatible char strings (included in one of its `of_device_id` structures) with the compatible property value of the DT device node. The process of associating a device with a device driver is called **binding**.

The `of_device_id` structure is declared in `include/linux/mod_devicetable.h` in the kernel source tree:

```

/*
 * Struct used for matching a device
 */
struct of_device_id {
    char    name[32];
    char    type[32];
    char    compatible[128];
    const void *data;
};

```

The main code sections of the driver will now be described:

1. Include the platform device header file which contains the structure and function declarations required by the platform devices/drivers:
2. Declare a list of devices supported by the driver. Create an array of structures of type `of_device_id`, and initialize their compatible fields with strings that will be used by the kernel to bind your driver with the devices declared in the Device Tree that include the same strings in their compatible properties. The platform bus driver will trigger the driver's `probe()` function if a match between device and driver occurs.

```

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,hellokeys"},
    {}
}

```

```
MODULE_DEVICE_TABLE(of, my_of_ids);
```

3. Add a `platform_driver` structure that will be registered with the platform bus:

```
static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "hellokeys",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};
```

4. After loading the kernel module, the function `my_probe()` will be called when a device matching one of the supported device ids is discovered. The function `my_remove()` will be called when the driver is unloaded. Therefore `my_probe()` does the role of the `hello_init()` function, and `my_remove()` does the role of the `hello_exit()` function. So, it makes sense to replace `hello_init()` with `my_probe()` and `hello_exit()` with `my_remove()`:

```
static int __init my_probe(struct platform_device *pdev)
{
    int ret_val;
    pr_info("my_probe() function is called.\n");
    ret_val = misc_register(&helloworld_miscdevice);
    if (ret_val != 0) {
        pr_err("could not register the misc device mydev");
        return ret_val;
    }
    pr_info("mydev: got minor %i\n", helloworld_miscdevice.minor);
    return 0;
}

static int __exit my_remove(struct platform_device *pdev)
{
    pr_info("my_remove() function is called.\n");
    misc_deregister(&helloworld_miscdevice);
    return 0;
}
```

5. Register your platform driver to the platform bus:

```
module_platform_driver(my_platform_driver);
```

6. Modify the Device Tree files (located in `arch/arm/boot/dts/` in the kernel source tree), including your DT device nodes. There must be a DT device node's `compatible` property identical to the compatible string stored in one of the driver's `of_device_id` structures. Open the `bcm2710-rpi-3-b.dts` DT file and add the `hellokeys` node in the `soc` node:

```
&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
```

```

        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };

    hellokeys {
        compatible = "arrow,hellokeys";
    };
}

```

7. Create a new `hellokeys_rpi3.c` file in the `linux_5.4_rpi3_drivers` folder, and add `hellokeys_rpi3.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```

~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy

```

8. Build the modified Device Tree, and load it to the target processor:

```

~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/

```

9. Reboot the Raspberry Pi:

```

root@raspberrypi:/home/pi# reboot

```

Listing 5-1: `hellokeys_rpi3.c`

```

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>

static int my_dev_open(struct inode *inode, struct file *file)
{
    pr_info("my_dev_open() is called.\n");
    return 0;
}

static int my_dev_close(struct inode *inode, struct file *file)
{
    pr_info("my_dev_close() is called.\n");
    return 0;
}

static long my_dev_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    pr_info("my_dev_ioctl() is called. cmd = %d, arg = %ld\n", cmd, arg);
    return 0;
}

static const struct file_operations my_dev_fops = {
    .owner = THIS_MODULE,
    .open = my_dev_open,

```

```

        .release = my_dev_close,
        .unlocked_ioctl = my_dev_ioctl,
};

static struct miscdevice helloworld_miscdevice = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "mydev",
    .fops = &my_dev_fops,
};

/* Add probe() function */
static int __init my_probe(struct platform_device *pdev)
{
    int ret_val;
    pr_info("my_probe() function is called.\n");
    ret_val = misc_register(&helloworld_miscdevice);

    if (ret_val != 0) {
        pr_err("could not register the misc device mydev");
        return ret_val;
    }

    pr_info("mydev: got minor %i\n", helloworld_miscdevice.minor);
    return 0;
}

/* Add remove() function */
static int __exit my_remove(struct platform_device *pdev)
{
    pr_info("my_remove() function is called.\n");
    misc_deregister(&helloworld_miscdevice);
    return 0;
}

/* Declare a list of devices supported by the driver */
static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,hellokeys"},
    {}},
};

MODULE_DEVICE_TABLE(of, my_of_ids);

/* Create a platform_driver structure */
static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "hellokeys",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};

```



```

/* Register your platform driver */
module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is the simplest platform driver");

```

hellokeys_rpi3.ko demonstration

See the Device Tree nodes under the root node:

```

root@raspberrypi:/home/pi# ls /proc/device-tree
'#address-cells'  cpus          model          soc
aliases          fixedregulator_3v3  name          __symbols__
arm-pmu          fixedregulator_5v0  __overrides__  system
axi              interrupt-parent  phy           thermal-zones
chosen           leds            reserved-memory  timer
clocks           memory@0        serial-number
compatible       memreserve      '#size-cells'

```

Check the Raspberry Pi model:

```

root@raspberrypi:/home/pi# cat /proc/device-tree/model
Raspberry Pi 3 Model B Rev 1.2

```

Load the module. The probe() function is called:

```

root@raspberrypi:/home/pi# insmod hellokeys_rpi3
my_probe() function is called.
mydev: got minor 60

```

See "hellokeys" sysfs entries:

```

root@raspberrypi:/home/pi# find /sys -name "*hellokeys*"
/sys/devices/platform/soc/soc:hellokeys
/sys/firmware/devicetree/base/soc/hellokeys
/sys/bus/platform/devices/soc:hellokeys
/sys/bus/platform/drivers/hellokeys
/sys/bus/platform/drivers/hellokeys/soc:hellokeys
/sys/module/hellokeys_rpi3
/sys/module/hellokeys_rpi3/drivers/platform:hellokeys

```

```

root@raspberrypi:/home/pi# ls -l /sys/bus/platform/drivers/hellokeys/
total 0
--w----- 1 root root 4096 Apr  8 18:48 bind
lrwxrwxrwx 1 root root    0 Apr  8 18:48 module -> ../../../../module/hellokeys_rpi3
lrwxrwxrwx 1 root root    0 Apr  8 18:48 soc:hellokeys -> ../../../../devices/platform/soc/
soc:hellokeys
--w----- 1 root root 4096 Apr  8 18:43 uevent
--w----- 1 root root 4096 Apr  8 18:48 unbind

```

```
root@raspberrypi:/home/pi# ls -l /sys/module/hellokeys_rpi3/drivers/
total 0
lrwxrwxrwx 1 root root 0 Apr  8 18:49 platform:hellokeys -> ../../../../bus/platform/drivers/hellokeys
```

Check that **mydev** is created under the **misc** class folder:

```
root@raspberrypi:/home/pi# ls /sys/class/misc/
autofs      cpu_dma_latency  hw_random      mydev        vscm-cma
cachefiles  fuse             loop-control   rfkill       watchdog
```

See the assigned mayor and minor numbers for the device **mydev**. The mayor number **10** is assigned by the **misc** framework:

```
root@raspberrypi:/home/pi# cat /sys/class/misc/mydev/dev
10:60
```

Run the application **ioctl_test**:

```
root@raspberrypi:/home/pi# ./ioctl_test
my_dev_open() is called.
my_dev_ioctl() is called. cmd = 100, arg = 110
my_dev_close() is called.
```

Remove the module:

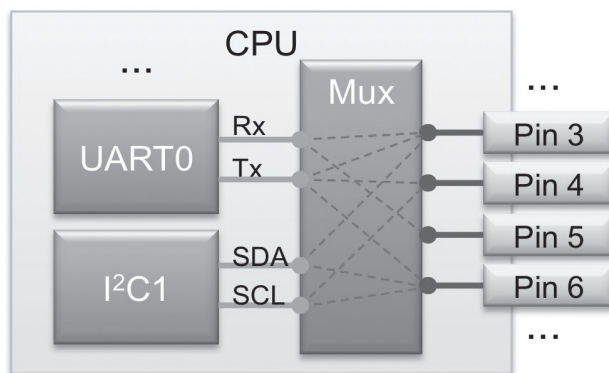
```
root@raspberrypi:/home/pi# rmmod hellokeys_rpi3.ko
Hello world exit
```

Documentation to interact with the hardware

In the following drivers, you will interact with some of the peripheral registers of the processor, so it will be necessary to download the Technical Reference Manual of the Raspberry Pi SoC to know its peripheral addresses. Go to the Raspberry Pi site (located at www.raspberrypi.org), and download the BCM2835 ARM Peripherals guide manual (located at <https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf>).

Hardware naming convention

A **pin** represents a physical input or output carrying an electrical signal. Every input or output signal goes through a physical pin from or into a component. Contact **pads** are designated surface areas of a printed circuit board or die of an integrated circuit. Some processors have a lot of functionality, but a limited number of pins (or pads). Even though a single pin can only perform one function at a time, it can be configured internally to perform different functions. This is called **pin multiplexing**. Every processor includes a Pin Controller, which enables that one pad can share several functional blocks. This sharing is done by multiplexing the pad's input and output signals, as shown in the following image:



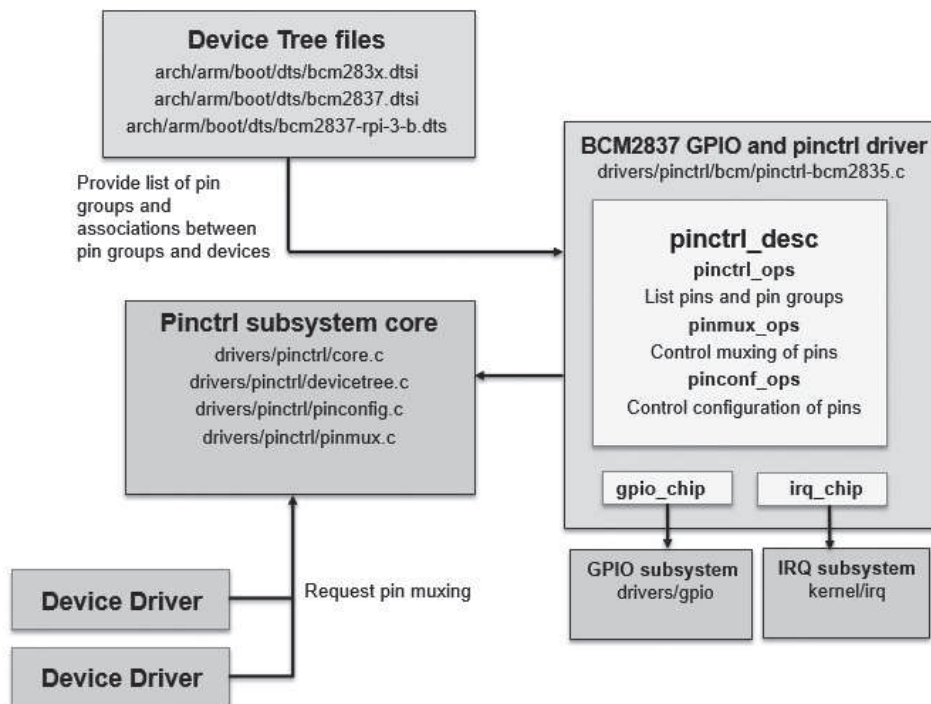
Pin control subsystem

In the old Linux pin muxing code, each architecture had its own pin-muxing code with a specific API. A lot of similar functionality was implemented in different ways. The pin-muxing had to be done at the SoC level, and it couldn't be requested by device drivers.

The **Pinctrl subsystem** aims at solving these problems. It is implemented in `drivers/pinctrl/` in the kernel source tree and provides:

- An API to register a pinctrl driver, for example, entities knowing the list of pins, their functions and how to configure them. This API is used by specific drivers of the SoC, for example, `pinctrl-bcm2835.c` (located in `drivers/pinctrl/bcm/` in the kernel source tree) to expose pin-muxing capabilities.
- An API for device drivers to request the muxing of a certain set of pins.
- Interaction with the GPIO drivers of the SoC.

Most pinctrl drivers provide a Device Tree binding, and the pin muxing must be described in the Device Tree. The exact Device Tree binding depends on each pinctrl driver. For the Broadcom BCM2835 GPIO (and pinmux) controller, the Device Tree binding is described in `Documentation/devicetree/bindings/pinctrl/brcm,bcm2835-gpio.txt` in the kernel source tree. In the next image, you can see the interaction of the BCM2837 pinctrl driver with the Device Tree and the Pinctrl, GPIO and IRQ subsystems:



The Pinctrl subsystem in Linux deals with:

- Enumerating and naming controllable pins.
- Multiplexing of pins. In the SoC, several pins can also form a pin group for a specific function. The pin control subsystem must manage all pin groups.
- Configuration of pins, such as software-controlled biasing, and driving mode specific pins, such as pull-up/down, open drain, load capacitance, etc.

All the pins of the BCM2837 SoC are named in the `pinctrl-bcm2835.c` `pinctrl` driver, as shown in the following code snippet. The `pinctrl_pin_desc` structure describes the pin name, pin index, etc.

```
/* pins are just named GPIO0..GPIO53 */
#define BCM2835_GPIO_PIN(a) PINCTRL_PIN(a, "gpio" #a)
static struct pinctrl_pin_desc bcm2835_gpio_pins[] = {
    BCM2835_GPIO_PIN(0),
    BCM2835_GPIO_PIN(1),
    BCM2835_GPIO_PIN(2),
    BCM2835_GPIO_PIN(3),
    BCM2835_GPIO_PIN(4),
    BCM2835_GPIO_PIN(5),
    BCM2835_GPIO_PIN(6),
    BCM2835_GPIO_PIN(7),
    BCM2835_GPIO_PIN(8),

    [...],

    BCM2835_GPIO_PIN(45),
    BCM2835_GPIO_PIN(46),
    BCM2835_GPIO_PIN(47),
    BCM2835_GPIO_PIN(48),
    BCM2835_GPIO_PIN(49),
    BCM2835_GPIO_PIN(50),
    BCM2835_GPIO_PIN(51),
    BCM2835_GPIO_PIN(52),
    BCM2835_GPIO_PIN(53),
};
```

The `pinctrl_dev` structure will be used to instantiate a Pin Controller and register a descriptor to the Pinctrl subsystem. This descriptor is represented by struct `pinctrl_desc`, which is used to describe the information of a Pin Controller. The `pinctrl_desc` structure (declared in `include/linux/pinctrl/pinctrl.h` in the kernel source tree) contains the pin definition, the pin configuration operation interface, the pin muxing operation interface, and the pin group operation interface supported by the Pin Controller of a SoC. The `pinctrl_desc` structure is declared as follows:

```
struct pinctrl_desc {
    const char *name;
    const struct pinctrl_pin_desc *pins;
    unsigned int npins;
    const struct pinctrl_ops *pctl_ops;
    const struct pinmux_ops *pmx_ops;
    const struct pinconf_ops *conf_ops;
    struct module *owner;
#ifdef CONFIG_GENERIC_PINCONF
    unsigned int num_custom_params;
    const struct pinconf_generic_params *custom_params;
    const struct pin_config_item *custom_conf_items;
#endif
    bool link_consumers;
};
```

Each physical pin is described through struct `pin_desc`. This structure is mainly used to record the use count of the pin and describes the current configuration information of the pin (function and group information the pin currently belongs to). The `pin_desc` structure is used by the `Pinctrl` subsystem to determine whether a pin is used in multiple configurations with different multiplexing conditions. The `pin_desc` structure is declared as follows:

```
/*
 * struct pin_desc - pin descriptor for each physical pin in the arch
 * @pctldev: corresponding pin control device
 * @name: a name for the pin, e.g. the name of the pin/pad/finger on a
 *        datasheet or such
 * @dynamic_name: if the name of this pin was dynamically allocated
 * @drv_data: driver-defined per-pin data. pinctrl core does not touch this
 * @mux_usecount: If zero, the pin is not claimed, and @owner should be NULL.
 * If non-zero, this pin is claimed by @owner. This field is an integer
 * rather than a boolean, since pinctrl_get() might process multiple
 * mapping table entries that refer to, and hence claim, the same group
 * or pin, and each of these will increment the @usecount.
 * @mux_owner: The name of device that called pinctrl_get().
 * @mux_setting: The most recent selected mux setting for this pin, if any.
 * @gpio_owner: If pinctrl_gpio_request() was called for this pin, this is
 * the name of the GPIO that "owns" this pin.
 */
struct pin_desc {
    struct pinctrl_dev *pctldev;
    const char *name;
    bool dynamic_name;
    void *drv_data;
    /* These fields only added when supporting pinmux drivers */
#ifdef CONFIG_PINMUX
    unsigned mux_usecount;
    const char *mux_owner;
    const struct pinctrl_setting_mux *mux_setting;
    const char *gpio_owner;
#endif
};
```

Following is a brief description of few important fields of the `pinctrl_desc` structure:

1. `struct pinctrl_ops`: Sometimes, it is necessary to combine several pins to achieve specific functions, such as SPI interface, I2C interface, etc. Therefore the Pin Controller needs to access and control multiple pins by group. The pin group is accessed and operated through the interface defined by `pinctrl_ops`. The `pinctrl_ops` structure is declared as follows:

```
/*
 * struct pinctrl_ops - global pin control operations, to be implemented by
 * pin controller drivers.
 * @get_groups_count: Returns the count of total number of groups registered.
 * @get_group_name: return the group name of the pin group
 * @get_group_pins: return an array of pins corresponding to a certain
 * group selector @pins, and the size of the array in @num_pins
```

```

* @pin_dbg_show: optional debugfs display hook that will provide per-device
*   info for a certain pin in debugfs
* @dt_node_to_map: parse a device tree "pin configuration node", and create
*   mapping table entries for it. These are returned through the @map and
*   @num_maps output parameters. This function is optional, and may be
*   omitted for pinctrl drivers that do not support device tree.
* @dt_free_map: free mapping table entries created via @dt_node_to_map. The
*   top-level @map pointer must be freed, along with any dynamically
*   allocated members of the mapping table entries themselves. This
*   function is optional, and may be omitted for pinctrl drivers that do
*   not support device tree.
*/
struct pinctrl_ops {
    int (*get_groups_count) (struct pinctrl_dev *pctldev);
    const char *(*get_group_name) (struct pinctrl_dev *pctldev,
                                   unsigned selector);
    int (*get_group_pins) (struct pinctrl_dev *pctldev,
                           unsigned selector,
                           const unsigned **pins,
                           unsigned *num_pins);
    void (*pin_dbg_show) (struct pinctrl_dev *pctldev, struct seq_file *s,
                           unsigned offset);
    int (*dt_node_to_map) (struct pinctrl_dev *pctldev,
                           struct device_node *np_config,
                           struct pinctrl_map **map, unsigned *num_maps);
    void (*dt_free_map) (struct pinctrl_dev *pctldev,
                          struct pinctrl_map *map, unsigned num_maps);
};

```

2. struct pinconf_ops: Pins can sometimes be configured in multiple ways by software, most of which are related to their electrical characteristics when used as input/output. For example, an output pin can be placed in a high-impedance state or "tri-state" (meaning it is effectively disconnected). You can connect an input pin to VDD or GND by setting a specific register value—pull-up/pull-down—so that there is a certain value on the pin when there is no signal to drive the pin or when it is not connected. The pinconf_ops structure is declared as follows:

```

/*
* struct pinconf_ops - pin config operations, to be implemented by
* pin configuration capable drivers.
* @is_generic: for pin controllers that want to use the generic interface,
*   this flag tells the framework that it's generic.
* @pin_config_get: get the config of a certain pin, if the requested config
*   is not available on this controller this should return -ENOTSUPP
*   and if it is available but disabled it should return -EINVAL
* @pin_config_set: configure an individual pin
* @pin_config_group_get: get configurations for an entire pin group; should
*   return -ENOTSUPP and -EINVAL using the same rules as pin_config_get.
* @pin_config_group_set: configure all pins in a group
* @pin_config_dbg_show: optional debugfs display hook that will provide
*   per-device info for a certain pin in debugfs
* @pin_config_group_dbg_show: optional debugfs display hook that will provide

```

```

*   per-device info for a certain group in debugfs
* @pin_config_config_dbg_show: optional debugfs display hook that will decode
*   and display a driver's pin configuration parameter
*/
struct pinconf_ops {
#ifdef CONFIG_GENERIC_PINCONF
    bool is_generic;
#endif
    int (*pin_config_get) (struct pinctrl_dev *pctldev,
                          unsigned pin,
                          unsigned long *config);
    int (*pin_config_set) (struct pinctrl_dev *pctldev,
                          unsigned pin,
                          unsigned long *configs,
                          unsigned num_configs);
    int (*pin_config_group_get) (struct pinctrl_dev *pctldev,
                                unsigned selector,
                                unsigned long *config);
    int (*pin_config_group_set) (struct pinctrl_dev *pctldev,
                                unsigned selector,
                                unsigned long *configs,
                                unsigned num_configs);
    void (*pin_config_dbg_show) (struct pinctrl_dev *pctldev,
                                struct seq_file *s,
                                unsigned offset);
    void (*pin_config_group_dbg_show) (struct pinctrl_dev *pctldev,
                                       struct seq_file *s,
                                       unsigned selector);
    void (*pin_config_config_dbg_show) (struct pinctrl_dev *pctldev,
                                       struct seq_file *s,
                                       unsigned long config);
};

```

3. struct pinmux_ops: Pinmux, also known as padmux or ballmux is a way for chip manufacturers to use a specific physical pin for multiple mutually exclusive functions. The SoC will contain several I2C, SPI, SDIO/MMC, and other peripheral controllers which can be routed to different pins through pin multiplexing settings. Because the number of GPIOs is often insufficient, many of the unused pins of the peripheral controllers are usually used as GPIO. The Pinctrl subsystem uses struct pinmux_ops to abstract pinmux related operations. The pinmux_ops structure is declared as follows:

```

/*
* struct pinmux_ops - pinmux operations, to be implemented by pin controller
* drivers that support pinmuxing
* @request: called by the core to see if a certain pin can be made
*   available for muxing. This is called by the core to acquire the pins
*   before selecting any actual mux setting across a function. The driver
*   is allowed to answer "no" by returning a negative error code
* @free: the reverse function of the request() callback, frees a pin after
*   being requested
* @get_functions_count: returns number of selectable named functions available
*   in this pinmux driver

```



```

* @get_function_name: return the function name of the muxing selector,
*   called by the core to figure out which mux setting it shall map a
*   certain device to
* @get_function_groups: return an array of groups names (in turn
*   referencing pins) connected to a certain function selector. The group
*   name can be used with the generic @pinctrl_ops to retrieve the
*   actual pins affected. The applicable groups will be returned in
*   @groups and the number of groups in @num_groups
* @set_mux: enable a certain muxing function with a certain pin group. The
*   driver does not need to figure out whether enabling this function
*   conflicts some other use of the pins in that group, such collisions
*   are handled by the pinmux subsystem. The @func_selector selects a
*   certain function whereas @group_selector selects a certain set of pins
*   to be used. On simple controllers the latter argument may be ignored
* @gpio_request_enable: requests and enables GPIO on a certain pin.
*   Implement this only if you can mux every pin individually as GPIO. The
*   affected GPIO range is passed along with an offset(pin number) into that
*   specific GPIO range - function selectors and pin groups are orthogonal
*   to this, the core will however make sure the pins do not collide.
* @gpio_disable_free: free up GPIO muxing on a certain pin, the reverse of
*   @gpio_request_enable
* @gpio_set_direction: Since controllers may need different configurations
*   depending on whether the GPIO is configured as input or output,
*   a direction selector function may be implemented as a backing
*   to the GPIO controllers that need pin muxing.
* @strict: do not allow simultaneous use of the same pin for GPIO and another
*   function. Check both gpio_owner and mux_owner strictly before approving
*   the pin request.
*/
struct pinmux_ops {
    int (*request) (struct pinctrl_dev *pctldev, unsigned offset);
    int (*free) (struct pinctrl_dev *pctldev, unsigned offset);
    int (*get_functions_count) (struct pinctrl_dev *pctldev);
    const char *(*get_function_name) (struct pinctrl_dev *pctldev,
                                     unsigned selector);
    int (*get_function_groups) (struct pinctrl_dev *pctldev,
                               unsigned selector,
                               const char * const **groups,
                               unsigned *num_groups);
    int (*set_mux) (struct pinctrl_dev *pctldev, unsigned func_selector,
                  unsigned group_selector);
    int (*gpio_request_enable) (struct pinctrl_dev *pctldev,
                              struct pinctrl_gpio_range *range,
                              unsigned offset);
    void (*gpio_disable_free) (struct pinctrl_dev *pctldev,
                              struct pinctrl_gpio_range *range,
                              unsigned offset);
    int (*gpio_set_direction) (struct pinctrl_dev *pctldev,
                              struct pinctrl_gpio_range *range,
                              unsigned offset,
                              bool input);
    bool strict;
};

```

The Pinctrl driver needs to implement the callback operations which are specific to the Pin Controller of the SoC. For example, the `pinctrl-bcm2835.c` pinctrl driver declares the following callback operations:

```
static struct pinctrl_desc bcm2835_pinctrl_desc = {
    .name = MODULE_NAME,
    .pins = bcm2835_gpio_pins,
    .npins = ARRAY_SIZE(bcm2835_gpio_pins),
    .pctlops = &bcm2835_pctl_ops,
    .pmxops = &bcm2835_pmx_ops,
    .confops = &bcm2835_pinconf_ops,
    .owner = THIS_MODULE,
};

static const struct pinctrl_ops bcm2835_pctl_ops = {
    .get_groups_count = bcm2835_pctl_get_groups_count,
    .get_group_name = bcm2835_pctl_get_group_name,
    .get_group_pins = bcm2835_pctl_get_group_pins,
    .pin_dbg_show = bcm2835_pctl_pin_dbg_show,
    .dt_node_to_map = bcm2835_pctl_dt_node_to_map,
    .dt_free_map = bcm2835_pctl_dt_free_map,
};

static const struct pinconf_ops bcm2835_pinconf_ops = {
    .is_generic = true,
    .pin_config_get = bcm2835_pinconf_get,
    .pin_config_set = bcm2835_pinconf_set,
};

static const struct pinmux_ops bcm2835_pmx_ops = {
    .free = bcm2835_pmx_free,
    .get_functions_count = bcm2835_pmx_get_functions_count,
    .get_function_name = bcm2835_pmx_get_function_name,
    .get_function_groups = bcm2835_pmx_get_function_groups,
    .set_mux = bcm2835_pmx_set,
    .gpio_disable_free = bcm2835_pmx_gpio_disable_free,
    .gpio_set_direction = bcm2835_pmx_gpio_set_direction,
};
```

Finally, `struct pinctrl_desc` is registered against the Pinctrl subsystem by calling the `devm_pinctrl_register()` function (defined in `drivers/pinctrl/core.c` in the kernel source tree), which is called in the `probe()` function of the pinctrl driver. In the following code snippet, extracted from the `pinctrl-bcm2835.c` pinctrl driver, you can see the registration of the `pinctrl_desc` for the Raspberry Pi device.

```
pc->pctl_dev = devm_pinctrl_register(dev, &bcm2835_pinctrl_desc, pc);
if (IS_ERR(pc->pctl_dev)) {
    gpiochip_remove(&pc->gpio_chip);
    return PTR_ERR(pc->pctl_dev);
}
```

The `devm_pinctrl_register()` function calls `pinctrl_register()`, which is mainly divided into two functions: `pinctrl_init_controller()` and `pinctrl_enable()`, as you can see in the following code snippet:

```
struct pinctrl_dev *pinctrl_register(struct pinctrl_desc *pctldesc,
                                   struct device *dev, void *driver_data)
{
    struct pinctrl_dev *pctldev;
    int error;

    pctldev = pinctrl_init_controller(pctldesc, dev, driver_data);
    if (IS_ERR(pctldev))
        return pctldev;

    error = pinctrl_enable(pctldev);
    if (error)
        return ERR_PTR(error);

    return pctldev;
}
```

The `pinctrl_init_controller()` function is used to describe the information of a Pin Controller, including the description of its supported pins, its pin multiplexing operation interfaces and its group related pin configuration interfaces. The `pinctrl_init_controller()` function will create a struct `pin_desc` for each pin and add it to the cardinal tree `pinctrl_dev->pin_desc_tree`. It will also check whether the callback operations in the `pinmux_ops`, `pinctrl_ops` and `pinconf_ops` variables are defined and do legality detection.

The `pinctrl_enable` function mainly adds the `pinctrl` device to the `pinctrldev_list`. At this time, the registration of the `pinctrl` device driver is basically completed.

GPIO controller driver

In the Linux kernel, the GPIO subsystem is implemented by GPIOlib. GPIOlib is a framework that provides an internal Linux kernel API for managing and configuring GPIOs, acting as a bridge between the Linux GPIO controller drivers and the Linux GPIO client drivers.

After years of development, the GPIOlib subsystem has produced two different mechanisms for gpio management, one is based on gpio num, and the other is based on gpio_desc descriptor form. The former mechanism has been abandoned due to various problems, but for backward compatibility, the kernel retains this mechanism, but for new gpio drivers is highly recommended to use the new interface.

Inside each GPIO controller driver, individual GPIOs are identified by their hardware number, which is a unique number between 0 and n, n being the number of GPIOs managed by the chip. This number is only internal to the driver. In addition to this internal number, each GPIO will also have a global number which can be used with the legacy GPIO interface. Each GPIO chip has a "base" number, and the GPIO global number will be calculated adding a hardware number to this base number. Although the integer representation is considered deprecated, it still has many users and thus needs to be maintained.

Each GPIO controller driver needs to include the header `linux/gpio/driver.h`, which defines the different structures that can be used to develop a GPIO controller driver.

The main structure of a GPIO controller driver is `struct gpio_chip` (declared in `include/linux/gpio/driver.h` in the kernel source tree), which includes GPIO operations specific to each GPIO controller of a SoC:

- Methods to establish GPIO line direction.
- Methods used to access GPIO line values.
- Method to set the electrical configuration of a given GPIO line.
- Method to return the IRQ number associated with a given GPIO line.
- Flag saying whether calls to its methods may sleep.
- Optional line names array to identify lines.
- Optional debugfs dump method (showing extra state like pullup config).
- Optional base number (will be automatically assigned if omitted).
- Optional label for diagnostics and GPIO chip mapping using platform data.

See below the GPIO operations for the BCM2837 SoC, extracted from the `pinctrl-bcm2835.c` pinctrl driver:

```
static const struct gpio_chip bcm2835_gpio_chip = {  
    .label = MODULE_NAME,  
    .owner = THIS_MODULE,  
    .request = gpiochip_generic_request,
```

```

        .free = gpiochip_generic_free,
        .direction_input = bcm2835_gpio_direction_input,
        .direction_output = bcm2835_gpio_direction_output,
        .get_direction = bcm2835_gpio_get_direction,
        .get = bcm2835_gpio_get,
        .set = bcm2835_gpio_set,
        .set_config = gpiochip_generic_config,
        .base = -1,
        .ngpio = BCM2835_NUM_GPIOS,
        .can_sleep = false,
    };

```

The code implementing a `gpio_chip` structure should support multiple instances of the controller by using the driver model. That code will configure each `gpio_chip` structure and issue `gpiochip_add_data()`. In the following code snippet, extracted from the `pinctrl-bcm2835.c` `pinctrl` driver, you can see the registration of the `gpio_chip` structure with the `GPIOlib` subsystem.

```

static int bcm2835_pinctrl_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct device_node *np = dev->of_node;
    struct bcm2835_pinctrl *pc;

    [...]

    pc->gpio_chip = bcm2835_gpio_chip;
    pc->gpio_chip.parent = dev;
    pc->gpio_chip.of_node = np;

    [...]

    err = gpiochip_add_data(&pc->gpio_chip, pc);
    if (err) {
        dev_err(dev, "could not add GPIO chip\n");
        return err;
    }

    [...]

    pc->gpio_range = bcm2835_pinctrl_gpio_range;
    pc->gpio_range.base = pc->gpio_chip.base;
    pc->gpio_range.gc = &pc->gpio_chip;
    pinctrl_add_gpio_range(pc->pctl_dev, &pc->gpio_range);

    return 0;
}

```

GPIO controllers can also provide interrupts, usually cascaded off a parent interrupt controller. The IRQ portions of the GPIO block are implemented via an `irq_chip` structure (declared in `include/linux/irq.h` in the kernel source tree). You will see a detailed description of the `irq_chip` structure in Chapter 7.

GPIO irqchips are usually included in one of three categories:

1. **CHAINED GPIO irqchips:** These are usually the type that is embedded on a SoC. This means that there is a fast IRQ flow handler for the GPIOs that gets called in a chain from the parent IRQ handler, most typically the system interrupt controller. This means that the GPIO irqchip handler will be called immediately from the parent irqchip while holding the IRQs disabled. The GPIO irqchip will then end up calling something like this sequence in its interrupt handler:

```
static irqreturn_t foo_gpio_irq(int irq, void *data)
{
    chained_irq_enter(...);
    generic_handle_irq(...);
    chained_irq_exit(...);
}
```

In the following code snippet, extracted from the `pinctrl-bcm2835.c` `pinctrl` driver, you can see the interrupt handler for the BCM2837 SoC:

```
static void bcm2835_gpio_irq_handler(struct irq_desc *desc)
{
    struct gpio_chip *chip = irq_desc_get_handler_data(desc);
    struct bcm2835_pinctrl *pc = gpiochip_get_data(chip);
    struct irq_chip *host_chip = irq_desc_get_chip(desc);
    int irq = irq_desc_get_irq(desc);
    int group, i;

    for (i = 0; i < BCM2835_NUM_IRQS; i++) {
        if (chip->irq_parents[i] == irq) {
            group = i;
            break;
        }
    }
    /* This should not happen, every IRQ has a bank */
    if (i == BCM2835_NUM_IRQS)
        BUG();

    chained_irq_enter(host_chip, desc);

    switch (group) {
    case 0: /* IRQ0 covers GPIOs 0-27 */
        bcm2835_gpio_irq_handle_bank(pc, 0, 0xffffffff);
        break;
    case 1: /* IRQ1 covers GPIOs 28-45 */
        bcm2835_gpio_irq_handle_bank(pc, 0, 0xf0000000);
        bcm2835_gpio_irq_handle_bank(pc, 1, 0x00003fff);
        break;
    case 2: /* IRQ2 covers GPIOs 46-53 */
        bcm2835_gpio_irq_handle_bank(pc, 1, 0x003fc000);
        break;
    }

    chained_irq_exit(host_chip, desc);
}
```

```
static void bcm2835_gpio_irq_handle_bank(struct bcm2835_pinctrl *pc,
                                         unsigned int bank, u32 mask)
{
    unsigned long events;
    unsigned offset;
    unsigned gpio;

    events = bcm2835_gpio_rd(pc, GPEDS0 + bank * 4);
    events &= mask;
    events &= pc->enabled_irq_map[bank];
    for_each_set_bit(offset, &events, 32) {
        gpio = (32 * bank) + offset;
        generic_handle_irq(irq_linear_revmap(pc->gpio_chip.irq.domain, gpio));
    }
}
```

2. **GENERIC CHAINED GPIO irqchips:** These are the same as "CHAINED GPIO irqchips", but chained IRQ handlers are not used. Instead, GPIO IRQs dispatching is performed by the generic IRQ handler, which is configured using `request_irq()`. The GPIO irqchip will then end up calling something like this sequence in its interrupt handler:

```
static irqreturn_t gpio_rcar_irq_handler(int irq, void *dev_id)
    for each detected GPIO IRQ
        generic_handle_irq(...);
```

3. **NESTED THREADED GPIO irqchips:** These are off-chip GPIO expanders and any other GPIO irqchip residing on the other side of a sleeping bus. Of course, such drivers that need slow bus traffic to read out the IRQ status and similar, traffic which may in turn incurs other IRQs to happen, cannot be handled in a quick IRQ handler with IRQs disabled. Instead, they need to spawn a thread and mask the parent IRQ line until the interrupt is handled by the driver. The hallmark of this driver is to call something like this in its interrupt handler:

```
static irqreturn_t foo_gpio_irq(int irq, void *data)
    [...]
    handle_nested_irq(irq);
```

In the LAB 7.4 of this book, you will implement a GPIO expander driver which includes a nested threaded GPIO irqchip.

GPIO descriptor consumer interface

This section describes the descriptor-based GPIO interface. For a description of the deprecated integer-based GPIO interface, please refer to `gpio-legacy.txt` under `Documentation/gpio/` folder.

All the functions that work with the descriptor-based GPIO interface are prefixed with **gpiod_**. The **gpio_** prefix is used for the legacy interface. No other function in the kernel should use these prefixes. The use of the legacy functions is strongly discouraged, new code should use `<linux/gpio/consumer.h>` and descriptors exclusively.

Obtaining and disposing GPIOs

The GPIO descriptor interface identifies each GPIO through a `gpio_desc` structure that is returned by using the `devm_gpiod_get()` function. This function takes as parameters: the GPIO consumer device (`dev`), the function within the GPIO consumer (`con_id`) and different optional GPIO initialization flags (`flags`):

```
struct gpio_desc *devm_gpiod_get(struct device *dev, const char *con_id,
                                enum gpiod_flags flags)
```

The `devm_gpiod_get_index()` variant of the `devm_gpiod_get()` function allows to access several GPIOs defined inside a specific GPIO function. The `devm_gpiod_get_index()` function returns the GPIO descriptor looking up the GPIO function (`con_id`) and its index (`idx`) from the Device Tree by using the `of_find_gpio()` function:

```
struct gpio_desc *devm_gpiod_get_index(struct device *dev,
                                       const char *con_id,
                                       unsigned int idx,
                                       enum gpiod_flags flags);
```

The `flags` parameter can specify a direction and initial value for the GPIO. These are some of its most significant values:

- `GPIO_ACTIVE_LOW` or 0 to not initialize the GPIO at all. The direction must be set later with one of the dedicated functions.
- `GPIO_ACTIVE_HIGH` to initialize the GPIO as input.
- `GPIO_DIRECTION_OUT` to initialize the GPIO as output with a value of 0.
- `GPIO_DIRECTION_OUT` to initialize the GPIO as output with a value of 1.

A GPIO descriptor can be disposed of by using the `devm_gpiod_put()` function.

Using GPIOs

Whenever you write a Linux driver that needs to control a GPIO, you must specify the GPIO direction. This can be done using the parameter `flags` of a `devm_gpiod_get*()` function or calling later a `gpiod_direction_*()` function if you have set the `flags` parameter to `GPIO_ACTIVE_LOW`:


```
int gpiod_direction_input(struct gpio_desc *desc);
int gpiod_direction_output(struct gpio_desc *desc, int value);
```

The return value is zero for success, else a negative errno. For output GPIOs, the value provided becomes the initial output value. This helps avoid signal glitching during system startup.

Most GPIO controllers can be accessed with memory read/write instructions. Those don't need to sleep and can safely be done from inside hard (non-threaded) IRQ handlers.

You can use the following functions to access GPIOs from an atomic context:

```
int gpiod_get_value(const struct gpio_desc *desc);
void gpiod_set_value(struct gpio_desc *desc, int value);
```

As a driver should not have to care about the physical line level, all of the `gpiod_set_value_XXX()` functions operate with the `*logical*` value. With this, they take the **active-low** property into account. This means that they check whether the GPIO is configured to be active-low, and if so, they manipulate the passed value before the physical line level is driven.

All the `gpiod_set_value_XXX()` functions interpret the parameter "value" as "active" ("1") or "inactive" ("0"). The physical line level will be driven accordingly.

As an example, if the active-low property for a dedicated GPIO is set and the `gpiod_set_value_XXX()` passes "active" ("1"), the physical line level will be driven low.

To summarize:

Function (example)	active-low property	physical line
<code>gpiod_set_value(desc, 0);</code>	default (active-high)	low
<code>gpiod_set_value(desc, 1);</code>	default (active-high)	high
<code>gpiod_set_value(desc, 0);</code>	active-low	high
<code>gpiod_set_value(desc, 1);</code>	active-low	low

GPIOs mapped to IRQs

An interrupt request can come in over a GPIO. You can get the Linux IRQ number corresponding to a given GPIO by using the following function:

```
int gpiod_to_irq(const struct gpio_desc *desc)
```

You will pass as a parameter the GPIO descriptor previously obtained using a `devm_gpiod_get*()` function, then the `gpiod_to_irq()` function will return the Linux IRQ number corresponding to that GPIO, or a negative errno code if the mapping can't be done. The `gpiod_to_irq()` function is not allowed to sleep.

Non-error values returned from `gpiod_to_irq()` can be passed to the `request_irq()` or `free_irq()` functions, which will request to obtain or release an interrupt. You will learn about these functions in the Chapter 7.

GPIOs in the Device Tree

GPIOs are mapped to devices and functions in the Device Tree. The exact way to do it depends on the GPIO controller providing the GPIOs (see the Device Tree bindings for your controller).

GPIOs mappings are defined in the consumer device node using a property named **<function>-gpios**, where **<function>** is requested by the Linux driver by calling the `gpiod_get()` function. For example:

```
foo_device {
    compatible = "acme,foo";
    ...
    led-gpios = <&gpioa 15 GPIO_ACTIVE_HIGH>, /* red */
               <&gpioa 16 GPIO_ACTIVE_HIGH>, /* green */
               <&gpioa 17 GPIO_ACTIVE_HIGH>; /* blue */

    power-gpios = <&gpiob 1 GPIO_ACTIVE_LOW>;
};
```

Where `&gpioa` and `&gpiob` are the handles to the specific gpio-controller nodes. Numbers 15, 16, 17 and 1 are the line offset for each gpio-controller, and `GPIO_ACTIVE_HIGH` is one of the flags used for the GPIO.

The `led-gpios` property will make GPIOs 15, 16 and 17 of the `gpioa` controller available to the Linux driver, and `power-gpios` will make GPIO 1 of the `gpiob` controller available to the driver:

```
struct gpio_desc *red, *green, *blue, *power;
red = gpiod_get_index(dev, "led", 0, GPIO_OUT_HIGH);
green = gpiod_get_index(dev, "led", 1, GPIO_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIO_OUT_HIGH);
power = gpiod_get(dev, "power", GPIO_OUT_HIGH);
```

The second parameter of the `gpiod_get*()` functions, the `con_id` string, has to be identical to the function prefix of the `gpios` suffixes used in the Device Tree. In the previous Device Tree example, you will use the next `con_id` parameters: "led" and "power" to obtain each GPIO descriptor from the `foo_device`. For the `led-gpios` property, in addition to the `con_id` parameter "led", you will also need to set the index (`idx`) with values 0, 1 or 2 in the `gpiod_get_index()` function to get each GPIO descriptor.

Exchanging data between kernel and user spaces

The Linux operating system prevents a user process from accessing another process and also prevents processes to access or manipulate directly the kernel data structures and services. This protection is achieved by separating the whole memory into two logical halves, the user and kernel space. The system calls are the fundamental interface between an application and the Linux kernel. The system calls are implemented in kernel space and their respective handlers are called through

APIs in user space. When a process executes a system call, the kernel will execute in process context on behalf of the calling process. When the kernel responds to interrupts, the interrupt handler runs asynchronously in interrupt context.

A driver for a device is the interface between an application and hardware. Accessing process address space from the kernel cannot be done directly (by de-referencing a user-space pointer). Direct access of a user space pointer can lead to incorrect behavior (depending on architecture, a user-space pointer may not be valid or mapped to kernel space), a kernel oops (the user-mode pointer can refer to a non-resident memory area) or security issues. Proper access to user space data is done by calling the macros/functions below:

1. A single value:

```
get_user(type val, type *address);
```

The val kernel variable gets the value pointed by the address user space pointer.

```
put_user(type val, type *address);
```

The value pointed by the address user space pointer is set to the contents of the val kernel variable.

2. A buffer:

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

copy_to_user() copies n bytes from the kernel space address pointed by from, to the user-space address pointed by to.

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

copy_from_user() copies n bytes from the user space address pointed by from, to the kernel space address pointed by to.

MMIO (Memory-Mapped I/O) device access

A peripheral device is controlled by writing and reading its registers. Often, a device has multiple registers that can be accessed at consecutive addresses either in the memory address space (MMIO) or in the I/O address space (PIO). See below the main differences between Port I/O and Memory-Mapped I/O:

1. MMIO

- Same address bus to address memory and I/O devices.
- Access to the I/O devices by using regular instructions.
- Most widely used I/O method across the different architectures supported by Linux.

2. PIO

- Different address spaces for memory and I/O devices.
- Uses a special class of CPU instructions to access I/O devices.

The BCM2837 SoC uses the MMIO access, so this method will be described in more detail during this section.

The Linux driver cannot access physical I/O addresses directly - **MMU mapping** is needed. For accessing I/O memory, drivers need to have a virtual address that the processor can handle, because I/O memory is not mapped by default in virtual memory.

You can obtain this I/O virtual address by using two different functions:

1. Map and remove mapping by using `ioremap()/iounmap()` functions. The `ioremap()` function accepts the physical address and the size of the area. It returns a pointer to virtual memory that can be dereferenced (or NULL if mapping is impossible).

```
void __iomem *ioremap(phys_addr_t offset, unsigned long size);
void iounmap(void *address);
```

2. Map and remove mapping attached to the driver device by using the `devm_ioremap()` and `devm_iounmap()` managed functions (declared in `include/linux/io.h` and defined in `lib/devres.c` in the kernel source tree), which simplify the driver code and error handling. Using `ioremap()` in device drivers is now deprecated.

```
void __iomem *devm_ioremap(struct device *dev, resource_size_t offset,
                          unsigned long size);
void devm_iounmap(struct device *dev, void __iomem *addr);
```

Each device (basic device structure) manages a linked list of resources via its included `list_head` `devres_head` structure. Calling a managed resource allocator involves adding the resource to the list. The resources are released in reverse order when the `probe()` function exits with an error status or after the `remove()` function returns. The use of managed functions in the `probe()` function removes the needed resource's releases on error handling, replacing `goto`'s and other resource's releases with just a `return`. It also removes resource's releases in the `remove()` function.

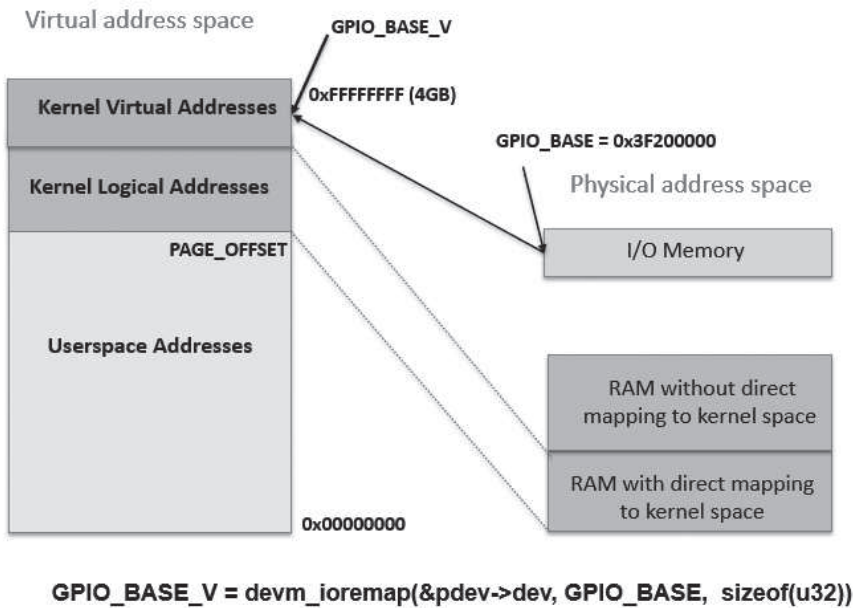
Dereferencing the pointer returned by `devm_ioremap()` is not reliable. Cache and synchronization issues may occur. The kernel provides functions to read and write to virtual addresses. To do PCI-style, little-endian accesses, conversion is being done automatically using the functions below:

```
unsigned read[bwl](void *addr);
void write[bwl](unsigned val, void *addr);
```

There are "generic" interfaces for doing new-style memory-mapped or PIO accesses. Architectures may do their own arch-optimized versions. These just act as wrappers around the old-style IO register access functions `read[bwl]/write[bwl]/in[bwl]/out[bwl]`:

```
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);
```

The following figure represents the GPIO base address (0x3F200000) mapping for the BCM2837 SoC. In the driver of the LAB 5.3, you can see how this register address is mapped to a virtual address by using the `devm_ioremap()` function.



LAB 5.2: "RGB LED platform device" module

In this lab, you will apply most of the concepts described so far during this chapter. You will control several LEDs, mapping from physical to virtual several peripheral register addresses of the SoC. You will create a character device per each LED by using the misc framework and control the

LEDs using `write()` and `read()` driver's operations. You will use the `copy_to_user()` and `copy_from_user()` functions to exchange character arrays between the kernel and user spaces.

LAB 5.2 hardware description

The BCM2837 SoC has 54 general-purpose I/O (GPIO) lines split into two banks. All the GPIO pins have at least two alternative functions within the SoC. The alternate functions are usually peripheral IO and a single peripheral may appear in each bank to allow flexibility on the choice of IO voltage. The GPIO has 41 registers. All accesses are assumed to be 32-bit.

The **function select registers** are used to define the operation of the general-purpose I/O pins. Each of the 54 GPIO pins has at least two alternative functions. The `FSEL{n}` field determines the functionality of the `n`th GPIO pin. All unused alternative function lines are tied to ground and will output a "0" if selected.

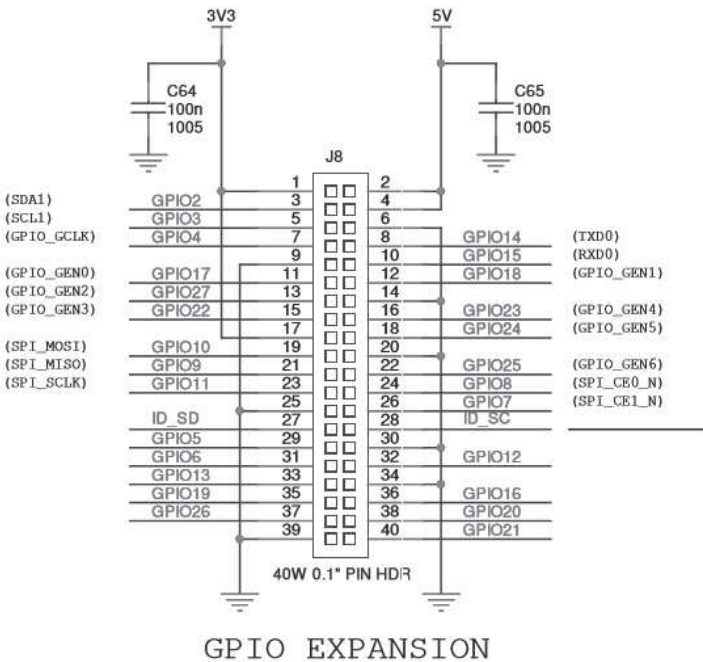
The **output set registers** are used to set a GPIO pin. The `SET{n}` field defines the respective GPIO pin to set, writing a "0" to the field has no effect. If the GPIO pin is being used as in input (by default) then the value in the `SET{n}` field is ignored. However, if the pin is subsequently defined as an output, then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations.

The **output clear registers** are used to clear a GPIO pin. The `CLR{n}` field defines the respective GPIO pin to clear, writing a "0" to the field has no effect. If the GPIO pin is being used as in input (by default) then the value in the `CLR{n}` field is ignored. However, if the pin is subsequently defined as an output, then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations.

For further information about GPIO pins available on BCM2837 SoC, you can see the section 6 General Purpose I/O (GPIO) of the BCM2835 ARM Peripherals guide [manual](#).

You will use three pins of the BCM2837 SoC to control several LEDs. These pins must be multiplexed as GPIOs in the Device Tree.

To obtain the GPIOs, you will use the Raspberry Pi 40-pin GPIO header, which is shown in the next figure:



To obtain the LEDs, you will use the Color click™ accessory board with mikroBUST™ form factor. See the Color click™ accessory board at <https://www.mikroe.com/color-click>. You can download the schematics from that link or from the GitHub repository of this book.

Connect the Raspberry Pi GPIO27 pin to the Color click™ RD pin, GPIO22 pin to GR and GPIO26 to BL. Connect 5V power and GND between both boards.

LAB 5.2 Device Tree description

As you have already seen in the previous Pin Controller section of this Chapter 5, the pin controller allows the processor to share one pad with several functional blocks. The sharing is done by multiplexing the PAD input/output signals. Since different modules require different PAD settings (e.g., pull up, keeper), the pin controller controls also the PAD settings parameters. Hardware modules whose signals are affected by pin configuration are designated "client devices". Again, each client device must be represented as a node in the Device Tree, just like any other hardware module. For a client device to operate correctly, certain pin controllers must set up certain specific pin configurations. Each pin controller must be represented as a node in the Device Tree. The pin controller of the BCM2837 SoC is declared in arch/arm/boot/dts/bcm283x.dtsi in the kernel source tree, as you can see in the following code snippet:

```

gpio: gpio@7e200000 {
    compatible = "brcm,bcm2835-gpio";
    reg = <0x7e200000 0xb4>;
    /*
     * The GPIO IP block is designed for 3 banks of GPIOs.
     * Each bank has a GPIO interrupt for itself.
     * There is an overall "any bank" interrupt.
     * In order, these are GIC interrupts 17, 18, 19, 20.
     * Since the BCM2835 only has 2 banks, the 2nd bank
     * interrupt output appears to be mirrored onto the
     * 3rd bank's interrupt signal.
     * So, a bank0 interrupt shows up on 17, 20, and
     * a bank1 interrupt shows up on 18, 19, 20!
     */
    interrupts = <2 17>, <2 18>, <2 19>, <2 20>;

    gpio-controller;
    #gpio-cells = <2>;

    interrupt-controller;
    #interrupt-cells = <2>;

    /* Defines common pin muxing groups
     *
     * While each pin can have its mux selected
     * for various functions individually, some
     * groups only make sense to switch to a
     * particular function together.
     */
    dpi_gpio0: dpi_gpio0 {
        brcm,pins = <0 1 2 3 4 5 6 7 8 9 10 11
                    12 13 14 15 16 17 18 19
                    20 21 22 23 24 25 26 27>;
        brcm,function = <BCM2835_FSEL_ALT2>;
    };
    emmc_gpio22: emmc_gpio22 {
        brcm,pins = <22 23 24 25 26 27>;
        brcm,function = <BCM2835_FSEL_ALT3>;
    };

    [...]

    i2c0_gpio0: i2c0_gpio0 {
        brcm,pins = <0 1>;
        brcm,function = <BCM2835_FSEL_ALT0>;
    };
    i2c0_gpio28: i2c0_gpio28 {
        brcm,pins = <28 29>;
        brcm,function = <BCM2835_FSEL_ALT0>;
    };

    [...]

    spi0_gpio7: spi0_gpio7 {

```



```

        brcm,pins = <7 8 9 10 11>;
        brcm,function = <BCM2835_FSEL_ALT0>;
    };
    spi0_gpio35: spi0_gpio35 {
        brcm,pins = <35 36 37 38 39>;
        brcm,function = <BCM2835_FSEL_ALT0>;
    };

    [...]

    uart1_gpio40: uart1_gpio40 {
        brcm,pins = <40 41>;
        brcm,function = <BCM2835_FSEL_ALT5>;
    };
    uart1_ctsrts_gpio42: uart1_ctsrts_gpio42 {
        brcm,pins = <42 43>;
        brcm,function = <BCM2835_FSEL_ALT5>;
    };
};

```

The compatible property value of the `gpio@7e200000` node matches with the compatible field value included in the `pinctrl-bcm2835.c` pinctrl driver.

The pin controller device should contain the pin configuration nodes for each client device. The contents of each of those pin configuration child nodes are defined entirely by the binding for the individual pin controller device. There exists no common standard for this content. Each pin configuration node lists the pin(s) to which it applies, one or more of the mux function to select on those pin(s), and pull-up/down configuration. These are the required properties for the pin controller of the BCM2837 SoC:

- `brcm,pins`: An array of cells. Each cell contains the ID of a pin. Valid IDs are the integer GPIO IDs; 0==GPIO0, 1==GPIO1, ... 53==GPIO53.
- `brcm,function`: Integer, containing the function to mux to the pin(s):
 - 0: GPIO in
 - 1: GPIO out
 - 2: alt5
 - 3: alt4
 - 4: alt0
 - 5: alt1
 - 6: alt2
 - 7: alt3
- `brcm,pull`: Integer, representing the pull-down/up to apply to the pin(s):
 - 0: none
 - 1: down
 - 2: up

Each of `brcm,function` and `brcm,pull` may contain either a single value which will be applied to all pins in `brcm,pins`, or 1 value for each entry in `brcm,pins`.

For further info, go to the Device Tree `pinctrl` documentation binding folder, located at `Documentation/devicetree/bindings/pinctrl/`, and examine the `brcm,bcm2835-gpio.txt` file.

For this lab, modify the `bcm2710-rpi-3-b.dts` Device Tree file by adding the `led_pins` pin configuration node in the `gpio` node and the `ledred`, `ledgreen` and `ledblue` device nodes in the SoC node:

```
/ {
    model = "Raspberry Pi 3 Model B";
};

&gpio {
    spi0_pins: spi0_pins {
        brcm,pins = <9 10 11>;
        brcm,function = <4>; /* alt0 */
    };

    spi0_cs_pins: spi0_cs_pins {
        brcm,pins = <8 7>;
        brcm,function = <1>; /* output */
    };

    i2c0_pins: i2c0 {
        brcm,pins = <0 1>;
        brcm,function = <4>;
    };

    [...]

    led_pins: led_pins {
        brcm,pins = <27 22 26>;
        brcm,function = <1>; /* Output */
        brcm,pull = <1 1 1>; /* Pull down */
    };
};

&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };

    hellokeys {
        compatible = "arrow,hellokeys";
    };
};
```

```

ledred {
    compatible = "arrow,RGBleds";
    label = "ledred";
    pinctrl-0 = <&led_pins>;
};

ledgreen {
    compatible = "arrow,RGBleds";
    label = "ledgreen";
};

ledblue {
    compatible = "arrow,RGBleds";
    label = "ledblue";
};

[...];

};

```

LAB 5.2 code description of the "RGB LED platform device" module

The main code sections of the driver will now be described:

1. Include the function headers:

```

#include <linux/module.h>
#include <linux/fs.h> /* struct file_operations */
#include <linux/platform_device.h> /* platform_driver_register(), platform_set_drvdata() */
#include <linux/io.h> /* devm_ioremap(), iowrite32() */
#include <linux/of.h> /* of_property_read_string() */
#include <linux/uaccess.h> /* copy_from_user(), copy_to_user() */
#include <linux/miscdevice.h> /* misc_register() */

```

2. Define the GPIO masks that will be used to configure the GPIO registers. See below the masks for the BCM2837 processor:

```

#define GPIO_27      27
#define GPIO_22      22
#define GPIO_26      26

/* To set and clear each individual LED */
#define GPIO_27_INDEX 1 << (GPIO_27 % 32)
#define GPIO_22_INDEX 1 << (GPIO_22 % 32)
#define GPIO_26_INDEX 1 << (GPIO_26 % 32)

/* Select the output function */
#define GPIO_27_FUNC 1 << ((GPIO_27 % 10) * 3)
#define GPIO_22_FUNC 1 << ((GPIO_22 % 10) * 3)
#define GPIO_26_FUNC 1 << ((GPIO_26 % 10) * 3)

```

```

/* Mask the GPIO functions */
/*
 * To select GPIO pin 27, you will set to 0b111 the bits 23-21 (FSEL7) of the
 * GPIO Alternate function select register 2. The GPIO pin 27 will control the red LED
 */
#define FSEL_27_MASK      0b111 << ((GPIO_27 % 10) * 3)

/*
 * To select GPIO pin 22, you will set to 0b111 the bits 8-6 (FSEL22) of the
 * GPIO Alternate function select register 2. The GPIO pin 22 will control the green LED
 */
#define FSEL_22_MASK      0b111 << ((GPIO_22 % 10) * 3)

/*
 * To select GPIO pin 26, you will set to 0b111 the bits 20-18 (FSEL26) of the
 * GPIO Alternate function select register 2. The GPIO pin 26 will control the blue LED
 */
#define FSEL_26_MASK      0b111 << ((GPIO_26 % 10) * 3) /* blue since bit 18 (FSEL26) */

#define GPIO_SET_FUNCTION_LEDS      (GPIO_27_FUNC | GPIO_22_FUNC | GPIO_26_FUNC)
#define GPIO_MASK_ALL_LEDS         (FSEL_27_MASK | FSEL_22_MASK | FSEL_26_MASK)
#define GPIO_SET_ALL_LEDS          (GPIO_27_INDEX | GPIO_22_INDEX | GPIO_26_INDEX)

```

3. Declare the physical I/O register addresses:

```

#define BCM2710_PERI_BASE      0x3F000000
#define GPIO_BASE              (BCM2710_PERI_BASE + 0x2000000)
#define GPFSEL2                GPIO_BASE + 0x08
#define GPSET0                  GPIO_BASE + 0x1C
#define GPCLR0                  GPIO_BASE + 0x28

```

4. Declare the `__iomem` pointers that will hold the virtual addresses returned by the `ioremap()` function:

```

static void __iomem *GPFSEL2_V;
static void __iomem *GPSET0_V;
static void __iomem *GPCLR0_V;

```

5. You will create an `led_dev` private structure to hold each device-specific information. In this driver, you will handle multiple char devices, so a `miscdevice` structure will be created for each device, then initialized and added to your device-specific data structure. The second field of the data structure is an `led_mask` variable that will hold a red, green or blue mask depending on the device. The last field of the private structure is a char array that will hold the command sent by the user space application to turn the LED on/off.

```

struct led_dev
{
    struct miscdevice led_misc_device; /* assign device for each led */
    u32 led_mask; /* different mask if led is R,G or B */

```

```

    const char *led_name; /* stores "label" string */
    char led_value[8];

};

```

6. Now, in your probe() routine, declare an instance of the private structure, and allocate it for each new probed device. The probe() function will be called three times (once per each DT node (ledred, ledgreen and ledblue) that matches its compatible property value with the compatible field value of the driver), allocating the corresponding devices.

```

    struct led_dev *led_device;
    led_device = devm_kzalloc(&pdev->dev, sizeof(struct led_dev), GFP_KERNEL);

```

7. In the led_init() routine, obtain the virtual addresses of the register addresses by using the ioremap() function, and store them in the __iomem pointers:

```

    GPFSSEL2_V = ioremap(GPFSSEL2, sizeof(u32));
    GPSET0_V = ioremap(GPSET0, sizeof(u32));
    GPCLR0_V = ioremap(GPCLR0, sizeof(u32));

```

8. Initialize each miscdevice structure within the probe() routine. As you have seen in the Chapter 4, the miscellaneous framework provides a simple layer above character files for devices that don't have any other framework to connect to. Registering with the misc subsystem simplifies the creation of a character file. The of_property_read_string() function will find and read a string from the label property of each probed DT device node. The third argument of the function is a pointer to a char pointer variable. The of_property_read_string() function will store the "label" string in the address pointed by the led_name pointer variable.

```

    of_property_read_string(pdev->dev.of_node, "label", &led_device->led_name);

    led_device->led_misc_device.minor = MISC_DYNAMIC_MINOR;
    led_device->led_misc_device.name = led_device->led_name;
    led_device->led_misc_device.fops = &led_fops;

```

9. When you are creating a character file, a file_operations structure is needed to define which driver functions to call when a user opens, closes, reads and writes to the char file. This structure will be stored in the miscdevice structure and passed to the misc subsystem when you register a device to it. One thing to note is that when you use the misc subsystem, it will automatically handle the "open" function for you. Inside the automatically created "open" function, it will tie the miscdevice structure to the private data field of the file that's been opened. This is useful to get access to the parent structure of the miscdevice structure by using the container_of() macro.

```

    static const struct file_operations led_fops = {
        .owner = THIS_MODULE,
        .read = led_read,
        .write = led_write,
    };

```

```
/* Pass file_operations structure to each created misc device */
led_device->led_misc_device.fops = &led_fops;
```

10. In the `probe()` routine, register each device with the kernel by using the `misc_register()` function. The `platform_set_drvdata()` function will attach each private structure to each `platform_device` structure. This will allow you to access your private data structure in other functions of the driver. You will recover the private structure in each `remove()` function call (called three times) by using the `platform_get_drvdata()` function:

```
ret_val = misc_register(&led_device->led_misc_device);
platform_set_drvdata(pdev, led_device);
```

11. Create the `led_write()` function that gets called whenever a write operation occurs on one of the character files. At the time you registered each misc device, you didn't keep any pointer to the `led_dev` structure. However, as the `miscdevice` structure is accessible through `file->private_data` and is a member of the `led_dev` structure, you can use a magic macro to compute the address of the parent structure; the `container_of()` macro gets the structure that `miscdevice` is stored inside of (which is your `led_dev` structure). The `copy_from_user()` function will get the on/off command from user space, then you will write to the corresponding register of the processor to switch on/off the LED using the `iowrite32()` function:

```
static ssize_t led_write(struct file *file, const char __user *buff,
                        size_t count, loff_t *ppos)
{
    const char *led_on = "on";
    const char *led_off = "off";
    struct led_dev *led_device;

    led_device = container_of(file->private_data,
                             struct led_dev, led_misc_device);

    if(copy_from_user(led_device->led_value, buff, count)) {
        pr_info("Bad copied value\n");
        return -EFAULT;
    }

    led_device->led_value[count-1] = '\0';

    if(!strcmp(led_device->led_value, led_on)) {
        iowrite32(led_device->led_mask, GPSET0_V);
    }
    else if (!strcmp(led_device->led_value, led_off)) {
        iowrite32(led_device->led_mask, GPCLR0_V);
    }
    else {
        pr_info("Bad value\n");
        return -EINVAL;
    }
}
```

```

    }

    return count;
}

```

12. Create the `led_read()` function that gets called whenever a read operation occurs on one of the character device files. You will use the `container_of()` macro to recover the private structure and the `copy_to_user()` function to return the `led_value` variable (on/off) to the user application:

```

static ssize_t led_read(struct file *file, char __user *buff,
                        size_t count, loff_t *ppos)
{
    struct led_dev *led_device;

    led_device = container_of(file->private_data, struct led_dev, led_misc_device);

    if(*ppos == 0) {
        if(copy_to_user(buff, &led_device->led_value,
                        sizeof(led_device->led_value))) {
            pr_info("Failed to return led_value to user space\n");
            return -EFAULT;
        }
        *ppos+=1;
        return sizeof(led_device->led_value);
    }

    return 0;
}

```

13. Declare a list of devices supported by the driver. Create an array of structures of type `of_device_id` where you initialize with strings the compatible fields that will be used by the kernel to bind your driver with the compatible Device Tree devices. This will automatically trigger your driver's `probe()` function if the Device Tree contains a compatible device entry (the probing happens three times in this driver).

```

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,RGBleds"},
    {}},
}
MODULE_DEVICE_TABLE(of, my_of_ids);

```

14. Add a `platform_driver` structure that will be registered to the platform bus:

```

static struct platform_driver led_platform_driver = {
    .probe = led_probe,
    .remove = led_remove,
    .driver = {
        .name = "RGBleds",
        .of_match_table = my_of_ids,
    }
}

```

```

        .owner = THIS_MODULE,
    };
};

```

15. In the `init()` function, register your driver with the platform bus driver by using the `platform_driver_register()` function:

```
platform_driver_register(&led_platform_driver);
```

16. Create a new `ledRGB_rpi3_platform.c` file in the `linux_5.4_rpi3_drivers` folder, and add `ledRGB_rpi3_platform.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

17. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

18. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 5-2: `ledRGB_rpi3_platform.c`

```

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/types.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/uaccess.h>
#include <linux/miscdevice.h>
#include <linux/delay.h>

struct led_dev
{
    struct miscdevice led_misc_device;
    u32 led_mask; /* different mask if led is R,G or B */
    const char *led_name;
    char led_value[8];
};

#define BCM2710_PERI_BASE    0x3F000000
#define GPIO_BASE            (BCM2710_PERI_BASE + 0x200000) /* GPIO controller */

#define GPIO_27              27
#define GPIO_22              22
#define GPIO_26              26

```



```

/* To switch on/off each LED */
#define GPIO_27_INDEX      1 << (GPIO_27 % 32)
#define GPIO_22_INDEX      1 << (GPIO_22 % 32)
#define GPIO_26_INDEX      1 << (GPIO_26 % 32)

/* Select the output function */
#define GPIO_27_FUNC        1 << ((GPIO_27 % 10) * 3)
#define GPIO_22_FUNC        1 << ((GPIO_22 % 10) * 3)
#define GPIO_26_FUNC        1 << ((GPIO_26 % 10) * 3)

/* Mask the GPIO functions */
#define FSEL_27_MASK        0b111 << ((GPIO_27 % 10) * 3) /* red since bit 21 (FSEL27) */
#define FSEL_22_MASK        0b111 << ((GPIO_22 % 10) * 3) /* green since bit 6 (FSEL22) */
#define FSEL_26_MASK        0b111 << ((GPIO_26 % 10) * 3) /* blue since bit 18 (FSEL26) */

#define GPIO_SET_FUNCTION_LEDS (GPIO_27_FUNC | GPIO_22_FUNC | GPIO_26_FUNC)
#define GPIO_MASK_ALL_LEDS     (FSEL_27_MASK | FSEL_22_MASK | FSEL_26_MASK)
#define GPIO_SET_ALL_LEDS      (GPIO_27_INDEX | GPIO_22_INDEX | GPIO_26_INDEX)

#define GPFSEL2              GPIO_BASE + 0x08
#define GPSET0               GPIO_BASE + 0x1C
#define GPCLR0               GPIO_BASE + 0x28

/* Declare __iomem pointers that will keep virtual addresses */
static void __iomem *GPFSEL2_V;
static void __iomem *GPSET0_V;
static void __iomem *GPCLR0_V;

static ssize_t led_write(struct file *file, const char __user *buff,
                        size_t count, loff_t *ppos)
{
    const char *led_on = "on";
    const char *led_off = "off";
    struct led_dev *led_device;

    pr_info("led_write() is called.\n");

    led_device = container_of(file->private_data, struct led_dev, led_misc_device);

    /*
     * In the command line "echo" add a \n character.
     * led_device->led_value = "on\n" or "off\n" after copy_from_user()
     */
    if(copy_from_user(led_device->led_value, buff, count)) {
        pr_info("Bad copied value\n");
        return -EFAULT;
    }

    /* Replace \n with \0 in led_device->led_value */
    led_device->led_value[count-1] = '\0';

    pr_info("This message received from User Space: %s", led_device->led_value);

    if(!strcmp(led_device->led_value, led_on)) {

```

```

        iowrite32(led_device->led_mask, GPSET0_V);
    }
    else if (!strcmp(led_device->led_value, led_off)) {
        iowrite32(led_device->led_mask, GPCLR0_V);
    }
    else {
        pr_info("Bad value\n");
        return -EINVAL;
    }

    pr_info("led_write() is exit.\n");
    return count;
}

static ssize_t led_read(struct file *file, char __user *buff, size_t count, loff_t *ppos)
{
    struct led_dev *led_device;

    pr_info("led_read() is called.\n");

    led_device = container_of(file->private_data, struct led_dev, led_misc_device);

    if(*ppos == 0){
        if(copy_to_user(buff, &led_device->led_value, sizeof(led_device->led_value))) {
            pr_info("Failed to return led_value to user space\n");
            return -EFAULT;
        }
        *ppos+=1;
        return sizeof(led_device->led_value);
    }

    pr_info("led_read() is exit.\n");

    return 0;
}

static const struct file_operations led_fops = {
    .owner = THIS_MODULE,
    .read = led_read,
    .write = led_write,
};

static int led_probe(struct platform_device *pdev)
{
    struct led_dev *led_device;
    int ret_val;
    char led_val[8] = "off\n";

    pr_info("leds_probe enter\n");

    led_device = devm_kzalloc(&pdev->dev, sizeof(struct led_dev), GFP_KERNEL);

    of_property_read_string(pdev->dev.of_node, "label", &led_device->led_name);
    led_device->led_misc_device.minor = MISC_DYNAMIC_MINOR;

```

```

    led_device->led_misc_device.name = led_device->led_name;
    led_device->led_misc_device.fops = &led_fops;

    if (strcmp(led_device->led_name, "ledred") == 0) {
        led_device->led_mask = GPIO_27_INDEX;
    }
    else if (strcmp(led_device->led_name, "ledgreen") == 0) {
        led_device->led_mask = GPIO_22_INDEX;
    }
    else if (strcmp(led_device->led_name, "ledblue") == 0) {
        led_device->led_mask = GPIO_26_INDEX;
    }
    else {
        pr_info("Bad device tree value\n");
        return -EINVAL;
    }

    /* Initialize the led status to off */
    memcpy(led_device->led_value, led_val, sizeof(led_val));

    ret_val = misc_register(&led_device->led_misc_device);
    if (ret_val) return ret_val; /* misc_register returns 0 if success */

    platform_set_drvdata(pdev, led_device);

    pr_info("leds_probe exit\n");

    return 0;
}

static int led_remove(struct platform_device *pdev)
{
    struct led_dev *led_device = platform_get_drvdata(pdev);

    pr_info("leds_remove enter\n");

    misc_deregister(&led_device->led_misc_device);

    pr_info("leds_remove exit\n");

    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,RGBleds"},
    {}
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver led_platform_driver = {
    .probe = led_probe,
    .remove = led_remove,
    .driver = {
        .name = "RGBleds",
    }
}

```

```

        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};

static int led_init(void)
{
    int ret_val;
    u32 GPFSEL_read, GPFSEL_write;
    pr_info("demo_init enter\n");

    ret_val = platform_driver_register(&led_platform_driver);
    if (ret_val != 0)
    {
        pr_err("platform value returned %d\n", ret_val);
        return ret_val;
    }

    GPFSEL2_V = ioremap(GPFSEL2, sizeof(u32));
    GPSET0_V = ioremap(GPSET0, sizeof(u32));
    GPCLR0_V = ioremap(GPCLR0, sizeof(u32));

    GPFSEL_read = ioread32(GPFSEL2_V); /* read current value */

    /*
     * Set FSEL27, FSEL26 and FSEL22 of GPFSEL2 register to 0,
     * keeping the value of the rest of GPFSEL2 bits,
     * then set to 1 the first bit of FSEL27, FSEL26 and FSEL22 to set
     * the direction of GPIO27, GPIO26 and GPIO22 to output
     */
    GPFSEL_write = (GPFSEL_read & ~GPIO_MASK_ALL_LEDS) | (GPIO_SET_FUNCTION_LEDS & GPIO_MASK_
ALL_LEDS);

    iowrite32(GPFSEL_write, GPFSEL2_V); /* set GPIOs to output */
    iowrite32(GPIO_SET_ALL_LEDS, GPCLR0_V); /* switch off all the leds, output is low */

    pr_info("demo_init exit\n");
    return 0;
}

static void led_exit(void)
{
    pr_info("led driver enter\n");
    iowrite32(GPIO_SET_ALL_LEDS, GPCLR0_V); /* switch off all the leds */

    iounmap(GPFSEL2_V);
    iounmap(GPSET0_V);
    iounmap(GPCLR0_V);

    platform_driver_unregister(&led_platform_driver);

    pr_info("led driver exit\n");
}

```

```

module_init(led_init);
module_exit(led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a platform driver that turns on/off three led devices");

```

ledRGB_rpi3_platform.ko demonstration

Load the module. The probe() function is called three times:

```

root@raspberrypi:/home/pi# insmod ledRGB_rpi3_platform.ko
demo_init enter
leds_probe enter
leds_probe exit
leds_probe enter
leds_probe exit
leds_probe enter
leds_probe exit
demo_init exit

```

See the led devices that have been created:

```

root@raspberrypi:/home/pi# ls /dev/led*
/dev/ledblue /dev/ledgreen /dev/ledred

```

Switch ON and OFF the LEDs and check their status:

```

root@raspberrypi:/home/pi# echo on > /dev/ledblue
root@raspberrypi:/home/pi# echo on > /dev/ledred
root@raspberrypi:/home/pi# echo on > /dev/ledgreen

root@raspberrypi:/home/pi# echo off > /dev/ledgreen
root@raspberrypi:/home/pi# echo off > /dev/ledred

root@raspberrypi:/home/pi# cat /dev/ledblue
root@raspberrypi:/home/pi# cat /dev/ledgreen

```

Remove the module. The remove() function is called three times:

```

root@raspberrypi:/home/pi# rmmod ledRGB_rpi3_platform.ko
led driver enter
leds_remove enter
leds_remove exit
leds_remove enter
leds_remove exit
leds_remove enter
leds_remove exit
led driver exit

```

Platform driver resources

Each device managed by a particular driver typically uses different hardware resources (for example, memory addresses for the I/O registers, DMA channels and IRQ lines).

The platform driver has access to the resources through a kernel API. These kernel functions automatically read standard platform device parameters from the struct `platform_device` resource array (declared in `include/linux/platform_device.h` in the kernel source tree). This resource array has been filled with the resource properties (for example, `reg`, `clocks` and `interrupts`) of the DT device nodes. In the `resource-names.txt` file, located in `Documentation/devicetree/bindings/` in the kernel source tree, you can see the different resource properties that can be accessed by index. The struct `platform_device` is declared as follows:

```
struct platform_device {
    const char *name;
    u32 id;
    struct device dev;
    u32 num_resources;
    struct resource *resource;
};
```

See below the declaration of the resource structure:

```
struct resource {
    resource_size_t start; /* unsigned int (resource_size_t) */
    resource_size_t end;
    const char *name;
    unsigned long flags;
    unsigned long desc;
    struct resource *parent, *sibling, *child;
};
```

This is the meaning of each field of the resource structure:

- `start/end`: This represents where the resource begins/ends. For I/O or memory regions, it represents where they begin/end. For IRQ lines, buses or DMA channels, `start/end` must have the same value.
- `flags`: This is a mask that characterizes the type of resource, for example `IORESOURCE_MEM`.
- `name`: This identifies or describes the resource.

There are a number of helper functions that get data out of the resource array:

1. **`platform_get_resource()`** -- Defined in `drivers/base/platform.c` in the kernel source tree, it gets a resource for a device and returns a resource structure filled with the DT values of the resource so that you can use these values later in the driver code, for example, to map them in the virtual space by using `devm_ioremap()` if they are physical memory addresses

(specified with `IORESOURCE_MEM` type). The `platform_get_resource()` function will check all the array resources until the type resource is found, then the resource structure is returned. See below the code of the `platform_get_resource()` function:

```
struct resource *platform_get_resource(struct platform_device *dev,
                                     unsigned int type,
                                     unsigned int num)
{
    int i;
    for (i = 0; i < dev->num_resources; i++) {
        struct resource *r = &dev->resource[i];
        if (type == resource_type(r) && num-- == 0)
            return r;
    }
    return NULL;
}
```

The first parameter of the previous function tells the function which device you are interested in so that it can extract the info needed. The second parameter depends on what kind of resource you are handling. If it is memory (or anything that can be mapped as memory), then it's `IORESOURCE_MEM`. You can see all the macros in `include/linux/ioport.h` in the kernel source tree. The last parameter determines which resource of that type is desired, with zero indicating the first one. Thus, for example, a driver could find and map its second MMIO region using the following lines of code:

```
struct resource *r;
r = platform_get_resource(pdev, IORESOURCE_MEM, 1);

/* ioremap your memory region */
g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));
```

The return value `r` is a pointer to the resource variable. The `resource_size()` function will return from the resource structure the memory size that will be mapped:

```
static inline resource_size_t resource_size(const struct resource *res)
{
    return res->end - res->start + 1;
}
```

2. **`platform_get_irq()`** -- This function extracts the resource structure from the `platform_device` structure, retrieving one of the interrupts properties declared in the Device Tree node. This function will be explained in more detail in the Chapter 7.

Linux LED class

The LED class will simplify the development of drivers that control LEDs. A "class" is both the implementation of a set of devices and the set of devices themselves. A class can be thought of as a

driver in the more general sense of the word. The device model has specific objects called "drivers" but a "class" is not one of those.

All the devices in a particular class tend to expose much the same interface, either to other devices or in user space (via sysfs or otherwise). Exactly how uniform the included devices are, is really up to the class though. Some aspects of the interfaces are optional, and not all devices implement them. It is not unheard-of for some devices in the same class to be completely different from others.

The LED class supports the blinking, flashing and brightness control features of physical LEDs. This class requires an underlying device to be available (`/sys/class/leds/<device>/`). This underlying device must be able to turn the LED on or off, may be able to set the brightness and might even provide timer functionality to autonomously blink the LED with a given period and duty cycle. Using the **brightness** file under each device subdirectory, the appropriate LED could be set to different brightness levels, for example, not just turned on and off but also dimmed. The data type used for passing the brightness level, enum `led_brightness`, defines only the levels `LED_OFF`, `LED_HALF` and `LED_FULL`:

```
enum led_brightness {
    LED_OFF = 0,
    LED_HALF = 127,
    LED_FULL = 255,
};
```

The LED class introduces the optional concept of LED trigger. A trigger is a kernel based source of LED events. The timer trigger is an example that will periodically change the LED brightness between `LED_OFF` and the current brightness setting. The "on" and "off" time can be specified via `/sys/class/leds/<device>/delay_{on,off}` sysfs entry in milliseconds. You can change the brightness value of an LED independently of the timer trigger. However, if you set the brightness value to `LED_OFF`, it will also disable the timer trigger.

A driver registering an LED class device will first allocate and fill an `led_classdev` structure (declared in `include/linux/leds.h` in the kernel source tree), then it will call `devm_led_classdev_register()`, which registers a new LED class object. The struct `led_classdev` is declared as follows:

```
struct led_classdev {
    const char          *name;
    enum led_brightness brightness;
    enum led_brightness max_brightness;

    [...]

    /*
     * Set LED brightness level. Use brightness_set_blocking for drivers
     * that can sleep while setting brightness.
     */
};
```



```

void (*brightness_set)(struct led_classdev *led_cdev, enum led_brightness brightness);
/*
 * Set LED brightness level immediately - it can block the caller for
 * the time required for accessing an LED device register.
 */
int (*brightness_set_blocking)(struct led_classdev *led_cdev,
                               enum led_brightness brightness);
/* Get LED brightness level */
enum led_brightness (*brightness_get)(struct led_classdev *led_cdev);

/*
 * Activate hardware accelerated blink, delays are in milliseconds
 * and if both are zero then a sensible default should be chosen.
 * The call should adjust the timings in that case and if it can't
 * match the values specified exactly.
 * Deactivate blinking again when the brightness is set to LED_OFF
 * via the brightness_set() callback.
 */
int (*blink_set)(struct led_classdev *led_cdev,
                 unsigned long *delay_on,
                 unsigned long *delay_off);

[...]

#ifdef CONFIG_LEDS_TRIGGERS
/* Protects the trigger data below */
struct rw_semaphore trigger_lock;

struct led_trigger      *trigger;
struct list_head trig_list;
void                    *trigger_data;
/* true if activated - deactivate routine uses it to do cleanup */
bool                    activated;
#endif

/* Ensures consistent access to the LED Flash Class device */
struct mutex led_access;
};

/*
 * devm_led_classdev_register - resource managed led_classdev_register()
 * @parent: The device to register.
 * @led_cdev: the led_classdev structure for this device.
 */
int devm_led_classdev_register(struct device *parent,
                              struct led_classdev *led_cdev)
{
    struct led_classdev **dr;
    int rc;

    dr = devres_alloc(devm_led_classdev_release, sizeof(*dr), GFP_KERNEL);
    if (!dr)
        return -ENOMEM;

```

```

rc = led_classdev_register(parent, led_cdev);
if (rc) {
    devres_free(dr);
    return rc;
}

*dr = led_cdev;
devres_add(parent, dr);

return 0;
}

```

LAB 5.3: "RGB LED class" module

In the previous LAB 5.2, you switched on/off several LEDs, creating a char device for each LED device by using the misc framework and writing to several GPIO registers. You used the write file operation and the `copy_from_user()` function to transmit a char array (on/off command) from user to kernel space.

In this LAB 5.3, you will use the LED subsystem to achieve something very similar to the LAB 5.2, but this time, you will simplify the code and add more functionalities, such as blinking each LED with a given period and duty cycle.

LAB 5.3 Device Tree description

In this LAB 5.3, you will keep the same DT GPIO multiplexing of the previous LAB 5.2, as you will use the same processor pads to control the LEDs. In the LAB 5.2, a DT device node was declared for each used LED, whereas in this LAB 5.3, you will declare a main LED RGB device node that includes several sub-nodes, each one representing an individual LED.

You will use in this new driver the `for_each_child_of_node()` function, which walks through the sub-nodes of the main node. Only the main node will have the compatible property, so after the matching between the device and the driver, the `probe()` function will be called only once, retrieving the info included in all the sub-nodes. The LED RGB device has a `reg` property that includes the GPIO register base address and the size of the address region it is assigned. After the driver and the device are matched, the `platform_get_resource()` function returns a resource structure filled with the values of the `reg` property so that you can use these values later in the driver code, mapping them in the virtual space by using the `devm_ioremap()` function.

Modify the `bcm2710-rpi-3-b.dts` Device Tree file by adding the following code in bold. The base address (`0x7e200000`) of the `reg` property is the `GPFSEL0` register address.

```

&soc {

```

```

virtgpio: virtgpio {
    compatible = "brcm,bcm2835-virtgpio";
    gpio-controller;
    #gpio-cells = <2>;
    firmware = <&firmware>;
    status = "okay";
};

[...]

ledclassRGB {
    compatible = "arrow,RGBclassleds";
    reg = <0x7e200000 0xb4>;
    pinctrl-names = "default";
    pinctrl-0 = <&led_pins>;

    red {
        label = "red";
    };

    green {
        label = "green";
    };

    blue {
        label = "blue";
        linux,default-trigger = "heartbeat";
    };
};
};

```

LAB 5.3 code description of the "RGB LED class" module

The main code sections of the driver will now be described:

1. Include the function headers:

```

#include <linux/module.h>
#include <linux/fs.h> /* struct file_operations */
#include <linux/platform_device.h> /* platform_driver_register(), platform_set_drvdata(),
platform_get_resource() */
#include <linux/io.h> /* devm_ioremap(), iowrite32() */
#include <linux/of.h> /* of_property_read_string() */
#include <linux/leds.h> /* misc_register() */

```

2. Define the GPIO masks that will be used to configure the GPIO registers of the SoC. You will take the base address stored in the DT reg property as a reference, then you will add an offset to it to set each register address. See below the masks for the BCM2837 processor:

```

#define GPIO_27      27
#define GPIO_22      22
#define GPIO_26      26

```

```

/* Offsets added to the base address */
#define GPFSEL2_offset 0x08
#define GPSET0_offset 0x1C
#define GPCR0_offset 0x28

/* Masks to switch on/off each LED */
#define GPIO_27_INDEX 1 << (GPIO_27 % 32)
#define GPIO_22_INDEX 1 << (GPIO_22 % 32)
#define GPIO_26_INDEX 1 << (GPIO_26 % 32)

/* Masks to select the output function */
#define GPIO_27_FUNC 1 << ((GPIO_27 % 10) * 3)
#define GPIO_22_FUNC 1 << ((GPIO_22 % 10) * 3)
#define GPIO_26_FUNC 1 << ((GPIO_26 % 10) * 3)

/*
 * To select GPIO pin 27, you will set to 0b111 the bits 23-21 (FSEL7) of the
 * GPIO Alternate function select register 2. The GPIO pin 27 will control the red LED
 */
#define FSEL_27_MASK 0b111 << ((GPIO_27 % 10) * 3)

/*
 * To select GPIO pin 22, you will set to 0b111 the bits 8-6 (FSEL22) of the
 * GPIO Alternate function select register 2. The GPIO pin 22 will control the green LED
 */
#define FSEL_22_MASK 0b111 << ((GPIO_22 % 10) * 3)

/*
 * To select GPIO pin 26, you will set to 0b111 the bits 20-18 (FSEL26) of the
 * GPIO Alternate function select register 2. The GPIO pin 26 will control the blue LED
 */
#define FSEL_26_MASK 0b111 << ((GPIO_26 % 10) * 3) /* blue since bit 18 (FSEL26) */
#define GPIO_SET_FUNCTION_LEDS (GPIO_27_FUNC | GPIO_22_FUNC | GPIO_26_FUNC)
#define GPIO_MASK_ALL_LEDS (FSEL_27_MASK | FSEL_22_MASK | FSEL_26_MASK)
#define GPIO_SET_ALL_LEDS (GPIO_27_INDEX | GPIO_22_INDEX | GPIO_26_INDEX)

```

3. You will create a private structure to hold the the specific data of the RGB LED device. In this driver, the first field of the private structure is an `led_mask` variable that will hold a red, green or blue mask depending of the LED device under control. The second field of the private structure is an `__iomem` pointer that will hold the GPIO register base address. The last field of the private structure is an `led_classdev` structure that will be initialized with some specific device settings. You will allocate a private structure for each sub-node device found.

```

struct led_dev
{
    u32 led_mask; /* different mask if led is R,G or B */
    void __iomem *base;
    struct led_classdev cdev;
};

```

4. See below a code snippet, extracted from the `probe()` routine, with the main lines of code marked in bold:

- The `platform_get_resource()` function gets the I/O registers resource described by the DT reg property.
- The `dev_ioremap()` function maps the area of register addresses to kernel virtual addresses.
- The `for_each_child_of_node()` function walks for each sub-node of the main node, allocating a private structure for each one by using the `devm_kzalloc()` function, then initializes the `led_classdev` field of each private structure.
- The `devm_led_classdev_register()` function registers each LED class device to the LED subsystem.

```
static int ledclass_probe(struct platform_device *pdev)
{
    void __iomem *g_ioremap_addr;
    struct device_node *child;
    struct resource *r;
    u32 GPFSEL_read, GPFSEL_write;
    struct device *dev = &pdev->dev;
    int ret = 0;
    int count;

    pr_info("platform_probe enter\n");

    /* Get our first memory resource from device tree */
    r = platform_get_resource(pdev, IORESOURCE_MEM, 0);

    /* ioremap our memory region */
    g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));

    [...]

    for_each_child_of_node(dev->of_node, child) {

        struct led_dev *led_device;
        struct led_classdev *cdev;
        led_device = devm_kzalloc(dev, sizeof(*led_device), GFP_KERNEL);

        cdev = &led_device->cdev;

        led_device->base = g_ioremap_addr;

        of_property_read_string(child, "label", &cdev->name);

        if (strcmp(cdev->name, "red") == 0) {
            led_device->led_mask = GPIO_27_INDEX;
        }
        else if (strcmp(cdev->name, "green") == 0) {
            led_device->led_mask = GPIO_22_INDEX;
        }
    }
}
```

```

    }
    else if (strcmp(cdev->name,"blue") == 0) {
        led_device->led_mask = GPIO_26_INDEX;
    }
    else {
        pr_info("Bad device tree value\n");
        return -EINVAL;
    }

    /* Disable timer trigger until led is on */
    led_device->cdev.brightness = LED_OFF;
    led_device->cdev.brightness_set = led_control;

    ret = devm_led_classdev_register(dev, &led_device->cdev);
}

pr_info("leds_probe exit\n");

return 0;
}

```

5. Write the `led_control()` function. Every time you write to the brightness sysfs entry (`/sys/class/leds/<device>/brightness`) under each device, the `led_control()` function is called. The LED subsystem hides the complexity of creating a class, the devices under the class, and the sysfs entries under each of the devices. Every time you write to the brightness sysfs entry, the `container_of()` function recovers the private structure associated with each device, then you can write to each register by using the `iowrite32()` function, which takes as a first parameter the `led_mask` value (stored in the private structure) associated to each LED. The following code snippet shows the `led_control()` function:

```

static void led_control(struct led_classdev *led_cdev, enum led_brightness b)
{
    struct led_dev *led = container_of(led_cdev, struct led_dev, cdev);

    iowrite32(GPIO_SET_ALL_LEDS, led->base + GPCLR0_offset);

    if (b != LED_OFF) /* LED ON */
        iowrite32(led->led_mask, led->base + GPSET0_offset);
    else
        iowrite32(led->led_mask, led->base + GPCLR0_offset); /* LED OFF */
}

```

6. Declare a list of devices supported by the driver:

```

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,RGBclassleds"},
    {}},
};

MODULE_DEVICE_TABLE(of, my_of_ids);

```

7. Add a `platform_driver` structure that will be registered to the platform bus:

```
static struct platform_driver led_platform_driver = {
    .probe = led_probe,
    .remove = led_remove,
    .driver = {
        .name = "RGBclassleds",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};
```

8. In the `init()` function, register your driver with the platform bus by using the `platform_driver_register()` function:

```
static int led_init(void)
{
    ret_val = platform_driver_register(&led_platform_driver);
    return 0;
}
```

9. Create a new `ledRGB_rpi3_class_platform.c` file in the `linux_5.4_rpi3_drivers` folder, and add `ledRGB_rpi3_class_platform.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

10. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

11. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 5-3: `ledRGB_rpi3_class_platform.c`

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/types.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/uaccess.h>
#include <linux/delay.h>
#include <linux/leds.h>

#define GPIO_27          27
#define GPIO_22          22
#define GPIO_26          26

#define GPFSEL2_offset   0x08
```

```

#define GPSET0_offset      0x1C
#define GPCR0_offset      0x28

/* Macros to switch on/off each LED */
#define GPIO_27_INDEX      1 << (GPIO_27 % 32)
#define GPIO_22_INDEX      1 << (GPIO_22 % 32)
#define GPIO_26_INDEX      1 << (GPIO_26 % 32)

/* Select the output function */
#define GPIO_27_FUNC        1 << ((GPIO_27 % 10) * 3)
#define GPIO_22_FUNC        1 << ((GPIO_22 % 10) * 3)
#define GPIO_26_FUNC        1 << ((GPIO_26 % 10) * 3)

#define FSEL_27_MASK        0b111 << ((GPIO_27 % 10) * 3) /* red since bit 21 (FSEL27) */
#define FSEL_22_MASK        0b111 << ((GPIO_22 % 10) * 3) /* green since bit 6 (FSEL22) */
#define FSEL_26_MASK        0b111 << ((GPIO_26 % 10) * 3) /* blue since bit 18 (FSEL26) */

#define GPIO_SET_FUNCTION_LEDS (GPIO_27_FUNC | GPIO_22_FUNC | GPIO_26_FUNC)
#define GPIO_MASK_ALL_LEDS    (FSEL_27_MASK | FSEL_22_MASK | FSEL_26_MASK)
#define GPIO_SET_ALL_LEDS    (GPIO_27_INDEX | GPIO_22_INDEX | GPIO_26_INDEX)

struct led_dev
{
    u32 led_mask; /* there are different masks if led is R,G or B */
    void __iomem *base;
    struct led_classdev cdev;
};

static void led_control(struct led_classdev *led_cdev, enum led_brightness b)
{
    struct led_dev *led = container_of(led_cdev, struct led_dev, cdev);

    iowrite32(GPIO_SET_ALL_LEDS, led->base + GPCR0_offset);

    if (b != LED_OFF) /* LED ON */
        iowrite32(led->led_mask, led->base + GPSET0_offset);
    else
        iowrite32(led->led_mask, led->base + GPCR0_offset); /* LED OFF */
}

static int ledclass_probe(struct platform_device *pdev)
{
    void __iomem *g_ioremap_addr;
    struct device_node *child;
    struct resource *r;
    u32 GPFSEL_read, GPFSEL_write;
    struct device *dev = &pdev->dev;
    int ret = 0;
    int count;

    pr_info("platform_probe enter\n");

    /* Get the first memory resource from the Device Tree */
    r = platform_get_resource(pdev, IORESOURCE_MEM, 0);

```



```

if (!r) {
    pr_err("IORESOURCE_MEM, 0 does not exist\n");
    return -EINVAL;
}
pr_info("r->start = 0x%08lx\n", (long unsigned int)r->start);
pr_info("r->end = 0x%08lx\n", (long unsigned int)r->end);

/* Map the memory region */
g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));
if (!g_ioremap_addr) {
    pr_err("ioremap failed \n");
    return -ENOMEM;
}

count = of_get_child_count(dev->of_node);
if (!count)
    return -EINVAL;

pr_info("there are %d nodes\n", count);

GPFSEL_read = ioread32(g_ioremap_addr + GPFSEL2_offset); /* read actual value */

GPFSEL_write = (GPFSEL_read & ~GPIO_MASK_ALL_LEDS) | (GPIO_SET_FUNCTION_LEDS & GPIO_MASK_
ALL_LEDS);

/* Set the direction of the GPIOs to output */
iowrite32(GPFSEL_write, g_ioremap_addr + GPFSEL2_offset);

/* Switch OFF all the LEDs */
iowrite32(GPIO_SET_ALL_LEDS, g_ioremap_addr + GPCLR0_offset);

for_each_child_of_node(dev->of_node, child) {

    struct led_dev *led_device;
    struct led_classdev *cdev;
    led_device = devm_kzalloc(dev, sizeof(*led_device), GFP_KERNEL);
    if (!led_device)
        return -ENOMEM;

    cdev = &led_device->cdev;

    led_device->base = g_ioremap_addr;

    of_property_read_string(child, "label", &cdev->name);

    if (strcmp(cdev->name, "red") == 0) {
        led_device->led_mask = GPIO_27_INDEX;
        //led_device->cdev.default_trigger = "heartbeat";
    }
    else if (strcmp(cdev->name, "green") == 0) {
        led_device->led_mask = GPIO_22_INDEX;
    }
    else if (strcmp(cdev->name, "blue") == 0) {
        led_device->led_mask = GPIO_26_INDEX;
    }
}

```

```

    }
    else {
        pr_info("Bad device tree value\n");
        return -EINVAL;
    }

    /* Disable the timer trigger */
    led_device->cdev.brightness = LED_OFF;
    led_device->cdev.brightness_set = led_control;

    ret = devm_led_classdev_register(dev, &led_device->cdev);
    if (ret) {
        dev_err(dev, "failed to register the led %s\n", cdev->name);
        of_node_put(child);
        return ret;
    }
}

pr_info("leds_probe exit\n");

return 0;
}

static int ledclass_remove(struct platform_device *pdev)
{
    pr_info("leds_remove enter\n");
    pr_info("leds_remove exit\n");

    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,RGBclassleds"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver led_platform_driver = {
    .probe = ledclass_probe,
    .remove = ledclass_remove,
    .driver = {
        .name = "RGBclassleds",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};

static int ledRGBclass_init(void)
{
    int ret_val;
    pr_info("demo_init enter\n");

    ret_val = platform_driver_register(&led_platform_driver);
    if (ret_val != 0)

```

```

    {
        pr_err("platform value returned %d\n", ret_val);
        return ret_val;
    }

    pr_info("demo_init exit\n");
    return 0;
}

static void ledRGBclass_exit(void)
{
    pr_info("led driver enter\n");

    platform_driver_unregister(&led_platform_driver);

    pr_info("led driver exit\n");
}

module_init(ledRGBclass_init);
module_exit(ledRGBclass_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a driver that turns on/off RGB leds using the LED subsystem");

```

ledRGB_rpi3_class_platform.ko demonstration

Load the module:

```

root@raspberrypi:/home/pi# insmod ledRGB_rpi3_class_platform.ko
demo_init enter
platform_probe enter
r->start = 0x3f200000
r->end = 0x3f2000b3
there are 3 nodes
leds_probe exit
demo_init exit

```

See the devices under the LED class:

```

root@raspberrypi:/home/pi# ls /sys/class/leds/
blue default-on green led0 led1 mmc0 red

```

Switch ON and OFF the LEDs, and blink the green LED:

```

root@raspberrypi:/home/pi# echo 1 > /sys/class/leds/red/brightness
root@raspberrypi:/home/pi# echo 1 > /sys/class/leds/blue/brightness
root@raspberrypi:/home/pi# echo 1 > /sys/class/leds/green/brightness

root@raspberrypi:/home/pi# cd /sys/class/leds/green
root@raspberrypi:/sys/class/leds/green# ls
brightness device max_brightness power subsystem trigger uevent
root@raspberrypi:/sys/class/leds/green# echo timer > trigger

```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod ledRGB_rpi3_class_platform.ko
led driver enter
leds_remove enter
leds_remove exit
led driver exit
```

Platform device drivers in user space

Device drivers in Linux, traditionally run in kernel space, but can also run in user space. It is not always necessary to write a device driver, especially when no two applications will require an exclusive access. The most useful example of this is a memory mapped device, but you can also do this with devices in the I/O space.

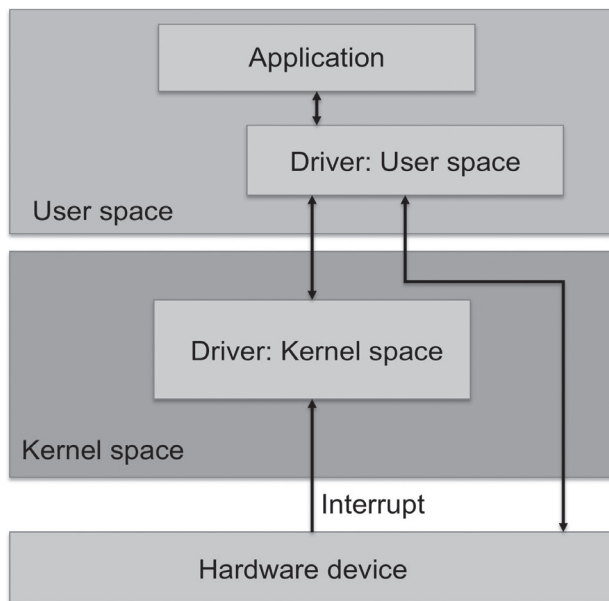
The Linux user space provides several advantages for device drivers, including more robust and flexible process management, standardized system call interface, simpler resource management, large number of libraries for XML or other configuration methods and regular expression parsing, among others. Each call to the kernel (system call) must perform a switch from user mode to supervisor mode and then back again. This takes time, which can become a performance bottleneck if the calls are frequent. It also makes applications more straightforward to debug by providing memory isolation and independent restart. At the same time, while kernel space applications need to conform to General Public License guidelines, user space applications are not bound by such restrictions.

On the other hand, user space drivers have their own drawbacks. Interrupt handling is the biggest challenge for a user space driver. The function handling an interrupt is called in privileged execution mode, often called supervisor mode. User space drivers have no permission to execute in privileged execution mode, making it impossible for user space drivers to implement an interrupt handler. To deal with this problem, you can perform polling or have a small kernel space driver handling only the interrupt. In the latter case, you can inform to the user space driver of an interrupt, either by a blocking call, which unblocks when an interrupt occurs, or using a POSIX signal to preempt the user space driver. If your driver must be accessible to multiple processes at once and/or manage contention for a resource, then you also need to write a real device driver at the kernel level, and a user space device driver will not be sufficient or even possible. Allocating memory that can be used for DMA transfers is also non-trivial for user space drivers. In kernel space, there are also frameworks that help solve device interdependencies.

The main advantages and disadvantages of using user space and kernel space drivers are summarized below:

1. User space driver advantages:
 - Easy to debug, as debug tools are more readily available for application development.
 - User space services such as floating point are available.
 - Device access is very efficient, as there is no system call required.
 - The application API on Linux is very stable.
 - The driver can be written in any language, not just C.
2. User space driver disadvantages:
 - No access to the kernel frameworks and services.
 - Interrupt handling cannot be done in user space. It must be handled by a kernel driver.
 - There is no predefined API to allow applications to access the device driver.
3. Kernel space driver advantages:
 - Run in kernel space in the highest privilege mode to allow access to interrupts and hardware resources.
 - There are a lot of kernel services such that kernel space drivers can be designed for complex devices.
 - The kernel provides an API to user space which allows multiple applications to access a kernel space driver simultaneously.
4. Kernel space driver disadvantages:
 - System call overhead to access drivers.
 - Challenging to debug.
 - Frequent kernel API changes. Kernel drivers built for one kernel version may not build for another.

The following image shows how a user space driver might be designed. The application interfaces to the user space part of the driver. The user space part handles the hardware, but uses its kernel space part for startup, shutdown and receiving interrupts.



User defined I/O: UIO

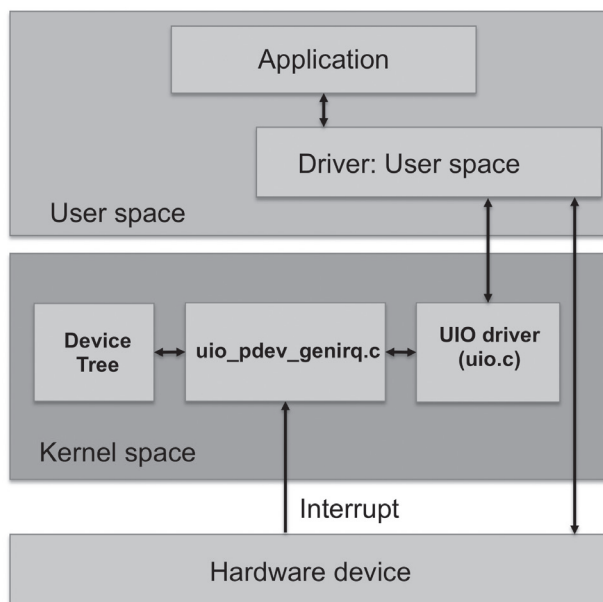
The Linux kernel provides a framework for developing user space drivers called **UIO**. This is a generic kernel driver which allows you to write user space drivers that are able to access device registers and handle interrupts.

There are two distinct UIO device drivers under `drivers/uio/` folder in the kernel source tree:

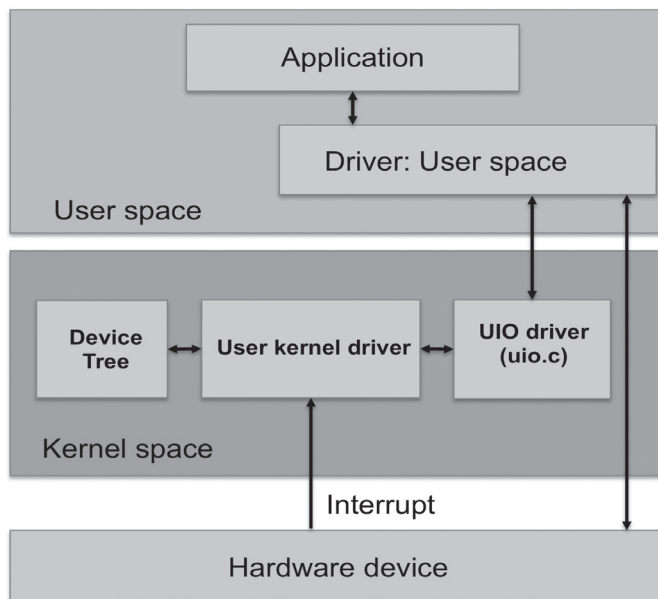
1. UIO driver (`drivers/uio.c`):
 - The UIO driver creates file attributes in the `sysfs`. These attributes describe the UIO device. The UIO driver also maps the device memory into the process address space using the `mmap()` function.
 - A minimal kernel space driver **`uio_pdrv_genirq`** ("UIO platform driver with generic interrupts") or **user provided kernel driver** is required to set up the UIO framework. The `uio.c` driver contains common support routines that are used by the `uio_pdrv_genirq.c` driver.

2. UIO platform device driver (drivers/uio_pdev_genirq.c):

- It provides the required kernel space driver for UIO.
- It works with the Device Tree. The Device Tree node needs to set "generic-uio" in its compatible property.
- The UIO platform device driver is configured from the Device Tree and registers a UIO device.



The UIO platform device driver can be replaced by a user provided kernel driver. The kernel space driver is a platform driver configured from the Device Tree that registers a UIO device inside the `probe()` function. The Device Tree node can use whatever you want in the compatible property, as it only has to match with the compatible string used in the kernel space driver, as with any other platform device driver.



The UIO drivers must be enabled in the kernel. Configure the kernel with `menuconfig`. Navigate from the **main menu** -> **Device Drivers** -> **Userspace I/O drivers**. Hit `<spacebar>` once to see a `<*>` appear next to the new configuration. Hit `<Exit>` until you exit the `menuconfig` GUI and remember to save the new configuration. In the Building the Linux kernel section of Chapter 1, you enabled the UIO drivers in the kernel, built the new kernel image and loaded it onto your Raspberry Pi.

How UIO works

Each UIO device is accessed through a device file and several `sysfs` attribute files. The device file will be called `/dev/uio0` for the first device, and `/dev/uio1`, `/dev/uio2` and so on for subsequent devices.

The UIO driver creates file attributes in the `sys` filesystem. The directory `/sys/class/uio/` is the root for all the file attributes. A separate numbered directory structure is created under `/sys/class/uio/` for each UIO device:

1. First UIO device: `/sys/class/uio/uio0`.
2. The `/sys/class/uio/uio0/name` directory contains the name of the device which correlates to the name in the struct `uio_info` structure.
3. The `/sys/class/uio/uio0/maps` directory has all the memory ranges for the device.
4. Each UIO device can make one or more memory regions available for memory mapping. Each mapping has its own directory in `sysfs`, the first mapping appears as `/sys/class/uio/uioX/maps/map0/`. Subsequent mappings create directories `map1/`, `map2/` and so on. These directories will only appear if the size of the mapping is not 0. Each `mapX/` directory contains four read-only files that show attributes of the memory:
 - **name:** A string identifier for this mapping. This is optional, the string can be empty. Drivers can set this to make it easier for user space to find the correct mapping.
 - **addr:** The address of memory that can be mapped.
 - **size:** The size, in bytes, of the memory pointed to by `addr`.
 - **offset:** The offset, in bytes, that has to be added to the pointer returned by `mmap()` to get to the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by `mmap()` are always page aligned, so it is a good practice to always add this offset.

Interrupts are handled by reading from `/dev/uioX`. A blocking `read()` from `/dev/uioX` will return as soon as an interrupt occurs. You can also use `select()` on `/dev/uioX` to wait for an interrupt. The integer value read from `/dev/uioX` represents the total interrupt count. You can use this number to figure out if you missed some interrupts.

The device memory is mapped into the process address space by calling the `mmap()` function of the UIO driver.

Kernel UIO API

The UIO API is small and simple to use:

1. The `uio_info` structure tells the framework the details of your driver. Some of the members are required, others are optional. These are some of the `uio_info` members:
 - **const char *name:** Required. The name of your driver as it will appear in `sysfs`. It is recommended to use the name of your module for this.
 - **const char *version:** Required. This string appears in `/sys/class/uio/uioX/version`.
 - **struct uio_mem mem[MAX_UIO_MAPS]:** Required if you have memory that can be mapped with `mmap()`. For each mapping you need to fill one of the `uio_mem` structures. See the description below for details.
 - **long irq:** Required. If your hardware generates an interrupt, it's your modules task to determine the `irq` number during initialization. If you don't have a hardware

generated interrupt but want to trigger the interrupt handler in some other way, set `irq` to `UIO_IRQ_CUSTOM`. If you had no interrupt at all, you could set `irq` to `UIO_IRQ_NONE`, though this rarely makes sense.

- **unsigned long `irq_flags`:** Required if you've set `irq` to a hardware interrupt number. The flags given here will be used in the call to `request_irq()`.
- **int (*`mmap`)(struct `uio_info` *`info`, struct `vm_area_struct` *`vma`):** Optional. If you need a special `mmap()` function, you can set it here. If this pointer is not `NULL`, your `mmap()` will be called instead of the built-in one.
- **int (*`open`)(struct `uio_info` *`info`, struct `inode` *`inode`):** Optional. You might want to have your own `open()`, e.g. to enable interrupts only when your device is actually used.
- **int (*`release`)(struct `uio_info` *`info`, struct `inode` *`inode`):** Optional. If you define your own `open()`, you will probably also want a custom `release()` function.
- **int (*`irqcontrol`)(struct `uio_info` *`info`, s32 `irq_on`):** Optional. If you need to be able to enable or disable interrupts from user space by writing to `/dev/uioX`, you can implement this function.

Usually, your device will have one or more memory regions that can be mapped to user space. For each region, you have to set up a `uio_mem` structure in the `mem[]` array. Here's a description of the fields of the `uio_mem` structure:

- **int `memtype`:** Required if mapping is used. Set this to `UIO_MEM_PHYS` if you have physical memory to be mapped. Use `UIO_MEM_LOGICAL` for logical memory (for example, allocated with `kmallocc()`). There's also `UIO_MEM_VIRTUAL` for virtual memory.
- **unsigned long `size`:** Fill in the size of the memory block that `addr` points to. If the size is zero, the mapping is considered unused. Note that you must initialize `size` with zero for all unused mappings.
- **void *`internal_addr`:** If you have to access this memory region from within your kernel module, you will want to map it internally by using something like `ioremap()`. Addresses returned by this function cannot be mapped to user space, so you must not store it in `addr`. Use `internal_addr` instead to remember such an address.

2. The function `uio_register_device()` connects the driver to the UIO framework:

- Requires a `uio_info` structure as an input.
- It is typically called from the `probe()` function of a platform device driver.
- It creates the device file `/dev/uio#` (`#` starting from 0) and all the associated `sysfs` file attributes.
- The function `uio_unregister_device()` disconnects the driver from the UIO framework, deleting the device file `/dev/uio#`.

LAB 5.4: "LED UIO platform" module

In this kernel module lab, you will develop a **UIO user space driver** that controls one of the LEDs used in the previous lab. The main function of a UIO driver is to expose the hardware registers to user space and does nothing within kernel space to control them. The LED will be controlled directly from the UIO user space driver by accessing to the memory mapped registers of the device. You will also write a **kernel driver** that obtains the register addresses from the Device Tree and initializes the `uio_info` structure with these values. You will also register the UIO device within the `probe()` function of the kernel driver.

LAB 5.4 Device Tree description

In this LAB 5.4, you will keep the same DT GPIO multiplexing of the previous LAB 5.3. In the LAB 5.3, you declared a main LED RGB device node that includes several sub-nodes, each one representing an individual LED. In this LAB 5.4, you will control only one LED, so you do not have to add any sub-node under the main node.

Modify the `bcm2710-rpi-3-b.dts` Device Tree file by adding the following code in bold. The base address (0x7e200000) of the `reg` property is the `GPFSEL0` register address.

```
&soc {

    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };

    [...]

    UIO {
        compatible = "arrow,UIO";
        reg = <0x7e200000 0x1000>;
        pinctrl-names = "default";
        pinctrl-0 = <&led_pins>;
    };

};
```

LAB 5.4 code description of the "LED UIO platform" module

The main code sections of the **user provided kernel driver** will now be described:

1. Include the function headers:

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

```
#include <linux/platform_device.h> /* platform_get_resource() */
#include <linux/io.h> /* devm_ioremap() */
#include <linux/uio_driver.h> /* struct uio_info, uio_register_device() */
```

2. Declare the `uio_info` structure:

```
static struct uio_info the_uio_info;
```

3. In the `probe()` function, the `platform_get_resource()` function returns the resource structure filled with the values described by the DT `reg` property. The `dev_ioremap()` function maps the area of register addresses to kernel virtual addresses:

```
struct resource *r;
void __iomem *g_ioremap_addr;

/* Get your first memory resource from device tree */
r = platform_get_resource(pdev, IORESOURCE_MEM, 0);

/* Map your memory region and get virtual address */
g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));
```

4. Initialize the `uio_info` structure:

```
the_uio_info.name = "led_uio";
the_uio_info.version = "1.0";
the_uio_info.mem[0].memtype = UIO_MEM_PHYS;
the_uio_info.mem[0].addr = r->start; /* physical address needed for the kernel user
mapping */
the_uio_info.mem[0].size = resource_size(r);
the_uio_info.mem[0].name = "demo_uio_driver_hw_region";
the_uio_info.mem[0].internal_addr = g_ioremap_addr; /* virtual address for internal
driver use */
```

5. Register the device to the UIO framework:

```
uio_register_device(&pdev->dev, &the_uio_info);
```

6. Declare a list of devices supported by the driver:

```
static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,UIO" },
    {} ,
};
MODULE_DEVICE_TABLE(of, my_of_ids);
```

7. Add a `platform_driver` structure that will be registered to the platform bus:

```
static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "UIO",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};
```

```
    }
};
```

8. Register your driver with the "Platform Driver" bus:

```
module_platform_driver(my_platform_driver);
```

9. Create a new `led_rpi3_UIO_platform.c` file in the `linux_5.4_rpi3_drivers` folder, and add `led_rpi3_UIO_platform.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

10. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

11. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

The main code sections of the **UIO user space driver** will now be described:

1. Include the function headers:

```
#include <sys/mman.h> /* mmap() */
```

2. Define the path to the sysfs parameter, from where you will obtain the size of memory that is going to be mapped:

```
#define UIO_SIZE "/sys/class/uio/uio0/maps/map0/size"
```

3. Open the UIO device:

```
open("/dev/uio0", O_RDWR | O_SYNC);
```

4. Obtain the memory size that is going to be mapped:

```
FILE *size_fp = fopen(UIO_SIZE, "r");
fscanf(size_fp, "%08X", &uio_size);
fclose(size_fp);
```

5. Perform mapping. A pointer to a virtual address will be returned, that corresponds to the **r->start** physical address obtained in the kernel space driver. You can now control the LED by writing to the virtual register address pointed to by the returned pointer variable. This user virtual address will be different to the kernel virtual address obtained with `dev_ioremap()` and pointed to by the `_uio_info.mem[0].internal_addr` variable.
6. In the apps folder, you will create the `UIO_app.c` file and write the Listing 5-5 code on it. Modify the Makefile file in the apps folder to compile and deploy the new application.

7. Compile the `UIO_app.c` application, and deploy it to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy.
```

Listing 5-4: `led_rpi3_UIO_platform.c`

```
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/io.h>
#include <linux/uio_driver.h>

static struct uio_info the_uio_info;

static int __init my_probe(struct platform_device *pdev)
{
    int ret_val;
    struct resource *r;
    struct device *dev = &pdev->dev;
    void __iomem *g_ioremap_addr;

    dev_info(dev, "platform_probe enter\n");

    /* Get the first memory resource from the Device Tree */
    r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!r) {
        dev_err(dev, "IORESOURCE_MEM, 0 does not exist\n");
        return -EINVAL;
    }
    dev_info(dev, "r->start = 0x%08lx\n", (long unsigned int)r->start);
    dev_info(dev, "r->end = 0x%08lx\n", (long unsigned int)r->end);

    /* ioremap the memory region and get the virtual address */
    g_ioremap_addr = devm_ioremap(dev, r->start, resource_size(r));
    if (!g_ioremap_addr) {
        dev_err(dev, "ioremap failed \n");
        return -ENOMEM;
    }

    /* Initialize uio_info struct uio_mem array */
    the_uio_info.name = "led_uio";
    the_uio_info.version = "1.0";
    the_uio_info.mem[0].memtype = UIO_MEM_PHYS;

    /* Physical address needed for the kernel user mapping */
    the_uio_info.mem[0].addr = r->start;
    the_uio_info.mem[0].size = resource_size(r);
    the_uio_info.mem[0].name = "demo_uio_driver_hw_region";

    /* Virtual address for internal driver use */
    the_uio_info.mem[0].internal_addr = g_ioremap_addr;

    /* Register the uio device */
```

```

    ret_val = uio_register_device(&pdev->dev, &the_uio_info);
    if (ret_val != 0) {
        dev_info(dev, "Could not register device \"led_uio\"...");
    }
    return 0;
}

static int __exit my_remove(struct platform_device *pdev)
{
    uio_unregister_device(&the_uio_info);
    dev_info(&pdev->dev, "platform_remove exit\n");

    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,UIO"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "UIO",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};
module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a UIO platform driver that turns the LED on/off \
    without using system calls");

```

Listing 5-5: UIO_app.c

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>

#define BUFFER_LENGTH      128

#define GPIO_27             27
#define GPIO_22             22
#define GPIO_26             26

```

```

#define GPFSEL2_offset      0x08
#define GPSET0_offset      0x1C
#define GPCLR0_offset      0x28

/* Masks to switch on/off each LED */
#define GPIO_27_INDEX      1 << (GPIO_27 % 32)
#define GPIO_22_INDEX      1 << (GPIO_22 % 32)
#define GPIO_26_INDEX      1 << (GPIO_26 % 32)

/* Masks to select the output function for each GPIO */
#define GPIO_27_FUNC        1 << ((GPIO_27 % 10) * 3)
#define GPIO_22_FUNC        1 << ((GPIO_22 % 10) * 3)
#define GPIO_26_FUNC        1 << ((GPIO_26 % 10) * 3)

#define FSEL_27_MASK        0b111 << ((GPIO_27 % 10) * 3) /* red since bit 21 (FSEL27) */
#define FSEL_22_MASK        0b111 << ((GPIO_22 % 10) * 3) /* green since bit 6 (FSEL22) */
#define FSEL_26_MASK        0b111 << ((GPIO_26 % 10) * 3) /* blue since bit 18 (FSEL26) */

#define GPIO_SET_FUNCTION_LEDS (GPIO_27_FUNC | GPIO_22_FUNC | GPIO_26_FUNC)
#define GPIO_MASK_ALL_LEDS    (FSEL_27_MASK | FSEL_22_MASK | FSEL_26_MASK)
#define GPIO_SET_ALL_LEDS     (GPIO_27_INDEX | GPIO_22_INDEX | GPIO_26_INDEX)

#define UIO_SIZE "/sys/class/uio/uio0/maps/map0/size"

int main()
{
    int ret, devuio_fd;
    int mem_fd;
    unsigned int uio_size;
    void *temp;
    int GPFSEL_read, GPFSEL_write;
    void *demo_driver_map;
    char sendstring[BUFFER LENGHT];
    char *led_on = "on";
    char *led_off = "off";
    char *Exit = "exit";

    printf("Starting led example\n");

    if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
        printf("can't open /dev/mem \n");
        exit(-1);
    }
    printf("opened /dev/mem \n");

    devuio_fd = open("/dev/uio0", O_RDWR | O_SYNC);
    if (devuio_fd < 0){
        perror("Failed to open the device");
        exit(EXIT_FAILURE);
    }
    printf("opened /dev/ui0 \n");

    /* Read the size that has to be mapped */

```



```

FILE *size_fp = fopen(UIO_SIZE, "r");
fscanf(size_fp, "0x%x", &uio_size);
fclose(size_fp);
printf("the value is %d\n", uio_size);

/* Do the mapping */
demo_driver_map = mmap(0, uio_size, PROT_READ | PROT_WRITE, MAP_SHARED, devuio_fd, 0);
if(demo_driver_map == MAP_FAILED) {
    perror("devuio mmap error");
    close(devuio_fd);
    exit(EXIT_FAILURE);
}

GPFSEL_read = *(int *)(demo_driver_map + GPFSEL2_offset);

GPFSEL_write = (GPFSEL_read & ~GPIO_MASK_ALL_LEDS) | (GPIO_SET_FUNCTION_LEDS & GPIO_MASK_
ALL_LEDS);

/* Set GPIO direction to output */
*(int *)(demo_driver_map + GPFSEL2_offset) = GPFSEL_write;

/* Switch off all the LEDs, output is low */
*(int *)(demo_driver_map + GPCLR0_offset) = GPIO_SET_ALL_LEDS;

/* Control the LED */
do {
    printf("Enter led value: on, off, or exit :\n");
    scanf("%[^\\n]%c", sendstring);

    if(strncmp(led_on, sendstring, 3) == 0)
    {
        temp = demo_driver_map + GPSET0_offset;
        *(int *)temp = GPIO_27_INDEX;
    }
    else if(strncmp(led_off, sendstring, 2) == 0)
    {
        temp = demo_driver_map + GPCLR0_offset;
        *(int *)temp = GPIO_27_INDEX;
    }
    else if(strncmp(Exit, sendstring, 4) == 0)
        printf("Exit application\n");
    else {
        printf("Bad value\n");
        return -EINVAL;
    }
} while(strncmp(sendstring, "exit", strlen(sendstring)));

ret = munmap(demo_driver_map, uio_size);
if(ret < 0) {
    perror("devuio munmap");
    close(devuio_fd);
    exit(EXIT_FAILURE);
}

```

```
    close(devuio_fd);  
    printf("Application terminated\n");  
    exit(EXIT_SUCCESS);  
}
```

led_rpi3_UIO_platform.ko with UIO_app demonstration

Load the module:

```
root@raspberrypi:/home/pi# insmod led_rpi3_UIO_platform.ko
```

Start the UIO_app application, which turns on/off an LED:

```
root@raspberrypi:/home/pi# ./UIO_app  
Starting led example  
Enter led value: on, off, or exit :  
on  
Enter led value: on, off, or exit :  
off  
Enter led value: on, off, or exit :  
exit  
Exit application  
Application terminated
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod led_rpi3_UIO_platform.ko
```

6

I2C Client Drivers

I2C is a protocol developed by Philips which original purpose was to link a CPU to other circuits in television sets. It is a two-wire, bi-directional serial bus for slow speed digital data that links one or more slave devices to a master (one or more bus controllers), providing a simple and efficient method of data transmission. The I2C protocol is widely used with embedded systems.

SMBus (System Management Bus) is a derivative developed by Intel based on the I2C protocol. The most common devices connected through SMBus are RAM modules configured using I2C EEPROMs and hardware monitoring chips that monitor critical parameters on PC motherboards and embedded systems. Because the SMBus is mostly a subset of the generalized I2C bus, you can use its protocols on many I2C systems. However, there are systems that don't meet both SMBus and I2C electrical constraints; and others which can't implement all the common SMBus protocol semantics or messages.

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

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

The previous routines read and write some bytes from/to a client. The first parameter is a pointer to struct i2c_client, which contains the I2C address of the client device. The second parameter is a pointer to the buffer that will store the data read from the slave and the data that will be written to the slave. The third parameter is the number of bytes to read/write (must be less than the length of the buffer, also should be less than 64k since msg.len is u16.). Returned is the actual number of bytes read/written.

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

The previous function sends a series of messages. Each message can be a read or write, and they can be mixed in any way. The transactions are combined: no stop bit is sent between transactions.

The `i2c_msg` structure contains for each message, the client address, the number of bytes of the message and the message data itself.

This is the generic function used to establish an SMBus communication:

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

All the functions below are implemented in terms of it. Never use the function `i2c_smbus_xfer()` directly.

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

You can see a detailed description of the SMBus functions at [Documentation/i2c/smbus-protocol.rst](#) in the kernel source tree.

The Linux I2C subsystem

The Linux I2C subsystem is based on the Linux device model and is composed of several drivers:

1. The **I2C bus core** of the I2C subsystem is located at `drivers/i2c/i2c-core.c` in the kernel source tree. In the device model, the I2C core is a collection of code that provides interface support between individual I2C client drivers and some I2C bus masters, such as the I2C controller drivers of the BCM2837 SoC. It manages bus arbitration, retry handling and various other protocol details. The I2C bus core is registered with the kernel using the `bus_register()` function and declares the I2C `bus_type` structure. The I2C core API is a set of functions used for an I2C client driver to send/receive data to/from a device connected to an I2C bus.
2. The **I2C controller drivers** are located in `drivers/i2c/busses/` in the kernel source tree. The I2C controller is a platform device that must be registered as a device to the platform bus. The I2C controller driver is a set of custom functions that issues read/writes to the I2C controller register addresses of the SoC. There is a specific code for each I2C controller on the SoC. These specific functions are called by the I2C core API when this invokes the `adap_algo_master_xfer` function after an I2C client driver has initiated an `i2c_transfer` function. In the I2C controller driver, you have to declare a private structure that includes an `i2c_adapter` structure. See below `struct bcm2835_i2c_dev` (declared in `drivers/i2c/busses/i2c-bcm2835.c` in the kernel source tree) for the BCM2837 SoC:

```
struct bcm2835_i2c_dev {
    struct device *dev;
    void __iomem *regs;
    int irq;
    struct i2c_adapter adapter;
    struct completion completion;
    struct i2c_msg *curr_msg;
    struct clk *bus_clk;
    int num_msgs;
    u32 msg_err;
    u8 *msg_buf;
    size_t msg_buf_remaining;
};
```

In the `probe()` function, this adapter structure is initialized for each I2C controller that has been probed:

```
adap = &i2c_dev->adapter;
i2c_set_adapdata(adap, i2c_dev);
adap->owner = THIS_MODULE;
adap->class = I2C_CLASS_DEPRECATED;
snprintf(adap->name, sizeof(adap->name), "bcm2835 (%s)",
         of_node_full_name(pdev->dev.of_node));
adap->algo = &bcm2835_i2c_algo;
```

```
adap->dev.parent = &pdev->dev;
adap->dev.of_node = pdev->dev.of_node;
```

The `bcm2835_i2c_algo` structure includes a pointer variable to the `bcm2835_i2c_xfer` function, which executes the specific code that will write/read the registers of the I2C hardware controller:

```
static const struct i2c_algorithm bcm2835_i2c_algo = {
    .master_xfer      = bcm2835_i2c_xfer,
    .functionality    = bcm2835_i2c_func,
};
```

Finally, in the `probe()` function, each I2C controller is added to the I2C bus core by calling the `i2c_add_numbered_adapter()` function (defined in `drivers/i2c/i2c-core.c`):

```
i2c_add_adapter(adap);
```

3. The **I2C device drivers** are located throughout `drivers/`, depending on the type of device (for example, `drivers/input/` for input devices). The driver code is specific to the device (for example, accelerometer, digital analog converter) and uses the I2C core API to send and receive data to/from the I2C device. For example, if the I2C client driver calls the `i2c_smbus_write_byte_data()` function (defined in `drivers/i2c/i2c-core.c`), you can see that this function calls the `i2c_smbus_xfer()` function:

```
s32 i2c_smbus_read_word_data(const struct i2c_client *client, u8 command)
{
    union i2c_smbus_data data;
    int status;

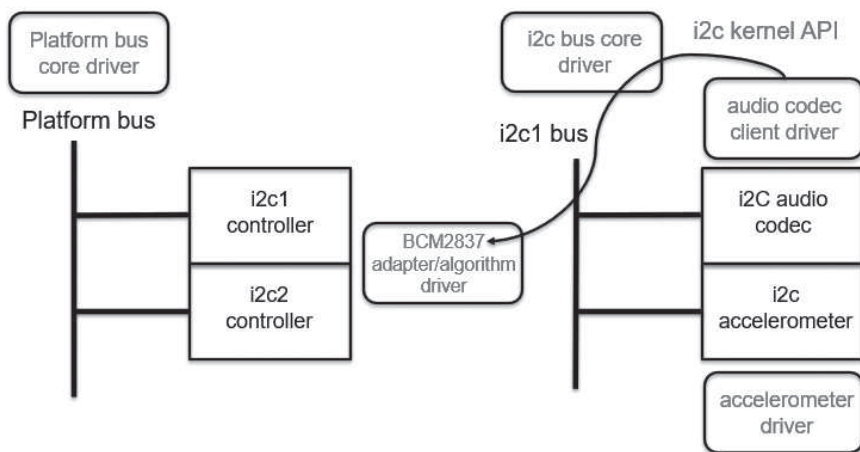
    status = i2c_smbus_xfer(client->adapter, client->addr, client->flags,
                           I2C_SMBUS_READ, command,
                           I2C_SMBUS_WORD_DATA, &data);
    return (status < 0) ? status : data.word;
}
```

If you check the code of the `i2c_smbus_xfer()` function, you can see that this function calls the `i2c_smbus_xfer_emulated()` function, which in turn calls the `master_xfer()` method:

```
i2c_smbus_read_word_data() -> i2c_smbus_xfer() -> i2c_smbus_xfer_emulated() -> i2c_
transfer() -> __i2c_transfer() -> bcm2835_i2c_xfer()
```

For the BCM2837 SoC, the `master_xfer` variable points to the `bcm2835_i2c_xfer()` function, which calls to the specific code that writes and reads the I2C controller registers.

See the I2C subsystem in the following figure. In the Linux device model, the `of_platform_populate()` function will register the I2C controller devices to the platform bus core. For the BCM2837 processor, the `i2c-bcm2835.c` controller driver registers itself to the platform bus core. The I2C client drivers are registered by themselves to the I2C bus core.



Writing I2C client drivers

You will now focus on writing I2C client drivers. In this and in successive chapters, you will develop several I2C client drivers that control I/O expanders, DACs, ADCs, accelerometers and Multidisplay LED controllers. In the following sections, you will see a description of the main steps to set up an I2C client driver.

I2C client driver registration

The I2C subsystem defines an `i2c_driver` structure (inherited from the `device_driver` structure) which must be instantiated and registered to the I2C bus core by each I2C device driver. Usually, you will implement a single driver structure and instantiate all clients from it. Remember, a driver structure contains general access routines and should be zero-initialized except for fields with data you provide. See below an example of an `i2c_driver` structure definition for an I2C accelerometer device:

```
static struct i2c_driver ioaccel_driver = {
    .driver = {
        .name = "mma8451",
        .owner = THIS_MODULE,
        .of_match_table = ioaccel_dt_ids,
    },
    .probe = ioaccel_probe,
    .remove = ioaccel_remove,
    .id_table = i2c_ids,
};
```

The `i2c_add_driver()` and `i2c_del_driver()` functions are used to register and unregister the driver. They are included in the `init()` and `exit()` functions. If the driver doesn't do anything else in these functions, use the `module_i2c_driver()` macro instead.

```
static int __init i2c_init(void)
{
    return i2c_add_driver(&ioaccel_driver);
}
module_init(i2c_init);

static void __exit i2c_cleanup(void)
{
    i2c_del_driver(&ioaccel_driver);
}
module_exit(i2c_cleanup);
```

In your device driver, create an array of `of_device_id` structures where you specify `.compatible` strings that should store the same value of the DT device node's compatible property. The `of_device_id` structure is declared in `include/linux/mod_devicetable.h` in the kernel source tree as follows:

```
struct of_device_id {
    char name[32];
    char type[32];
    char compatible[128];
};
```

The `of_match_table` field (included in the driver field) of the `i2c_driver` structure is a pointer to the array of `of_device_id` structures that hold the compatible strings supported by the driver:

```
static const struct of_device_id ioaccel_dt_ids[] = {
    { .compatible = "fsl,mma8451", },
    { }
};
MODULE_DEVICE_TABLE(of, ioaccel_dt_ids);
```

The driver's `probe()` function is called when the compatible field in one of the `of_device_id` entries matches with the compatible property of a DT device node. The `probe()` function is responsible for initializing the device with the configuration values obtained from the matched DT device node and also to register the device to the appropriate kernel framework.

In your I2C device driver, you also have to define an array of `i2c_device_id` structures. This array is pointed to by the `id_table` field of `struct i2c_driver`. The `id_table` is used for non-DT based probing of I2C devices. If your driver only uses DT based probing, then you can use the `probe_new()` function instead of the `probe()` one.

```
static const struct i2c_device_id mma8451_id[] = {
    { "mma8450", 0 },
    { "mma8451", 1 },
    { }
};
```



```
MODULE_DEVICE_TABLE(i2c, mma8451_id);
```

The second argument of the `probe()` function is an element of this array related to your attached device:

```
static ioaccel_probe(struct i2c_client *client, const struct i2c_device_id *id);
```

The binding will happen based on the `i2c_device_id` table or Device Tree compatible string. The I2C core first tries to match the device by compatible string (OF style, which is Device Tree), and if it fails, it then tries to match the device using the `id_table`.

Declaration of I2C devices in the Device Tree

In the Device Tree, an I2C controller is typically declared in the `.dtsi` file that describes the processor (for the BCM2837 SoC, see `arch/arm/boot/dts/bcm283x.dtsi`). The I2C controller definition is normally declared with `status = "disabled"`. For example, in the `bcm283x.dtsi` file, there are declared three I2C controllers that will be registered to the I2C bus core through the `of_platform_populate()` function. For the BCM2837 SoC, the `i2c-bcm2835.c` driver will register itself to the I2C bus core by using the `module_i2c_driver()` function. The `probe()` function will be called three times (one for each compatible = "brcm,bcm2835-i2c" matching), initializing an `i2c_adapter` structure for each controller and registering it with the I2C bus core by using the `i2c_add_adapter()` function. See below the declaration of the I2C controller nodes in the Device Tree for the BCM2837 SoC:

```
i2c0: i2c@7e205000 {
    compatible = "brcm,bcm2835-i2c";
    reg = <0x7e205000 0x200>;
    interrupts = <2 21>;
    clocks = <&clocks BCM2835_CLOCK_VPU>;
    #address-cells = <1>;
    #size-cells = <0>;
    status = "disabled";
};
i2c1: i2c@7e804000 {
    compatible = "brcm,bcm2835-i2c";
    reg = <0x7e804000 0x1000>;
    interrupts = <2 21>;
    clocks = <&clocks BCM2835_CLOCK_VPU>;
    #address-cells = <1>;
    #size-cells = <0>;
    status = "disabled";
};
i2c2: i2c@7e805000 {
    compatible = "brcm,bcm2835-i2c";
    reg = <0x7e805000 0x1000>;
    interrupts = <2 21>;
    clocks = <&clocks BCM2835_CLOCK_VPU>;
    #address-cells = <1>;
    #size-cells = <0>;
```

```
    status = "disabled";
};
```

The Device Tree declaration for I2C devices is done as sub-nodes of the master controller at the board/platform level (arch/arm/boot/dts/bcm2710-rpi-3-b.dts):

- The I2C controller device is enabled (status = "okay").
- The I2C bus frequency is defined by using the clock-frequency property.
- The I2C devices on the bus are described as children of the I2C controller node. The reg property provides the I2C slave address on the bus.
- In the I2C device node, check that the compatible property matches with one of the driver's of_device_id compatible strings.

LAB 6.1: "I2C I/O expander device" module

Throughout the upcoming lab, you will implement your first driver to control an I2C device. The driver will manage a PCF8574 I/O expander device connected to the I2C bus. You can use one the multiple boards based on this device to develop this lab, for example, the following one:

<https://www.waveshare.com/pcf8574-io-expansion-board.htm>.

LAB 6.1 hardware description

You will use the Raspberry Pi GPIO expansion connector to obtain the I2C signals. The GPIO2 and GPIO3 pins will be used to get the SDA1 and SCL1 signals.

LAB 6.1 Device Tree description

Open and modify the bcm2710-rpi-3-b.dts Device Tree file by adding the ioexp@39 sub-node inside the i2c1 controller master node. The i2c1 controller is enabled by writing "okay" to the status property. The pinctrl-0 property of the i2c1 node points to the i2c1_pins pin configuration node, where the GPIO2 and GPIO3 pads are multiplexed as I2C signals. In the declaration of the sub-node device, the reg property provides the I2C address of the PCF8574 I/O expander connected to the I2C bus.

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
    status = "okay";

    [...]

    ioexp@39 {
        compatible = "arrow,ioexp";
        reg = <0x39>;
```

```
};  
}
```

See below the `i2c1_pins` pin configuration node, where the I2C controller pads are multiplexed as I2C signals:

```
i2c1_pins: i2c1 {  
    brcm,pins = <2 3>; /* GPIO2 and GPIO3 pins */  
    brcm,function = <4>; /* ALT0 mux function */  
};
```

You can see that the GPIO2 and GPIO3 pins are set to the ALT0 function. See the meaning of `brcm,function = <4>` in the `brcm,bcm2835-gpio.txt` document located in `Documentation/devicetree/bindings/pinctrl/` in the kernel source tree.

Open the file `BCM2835 ARM Peripherals` guide, and find the table included in the section 6.2 Alternative Function Assignments. You can see in the following screenshot that GPIO2 and GPIO3 pins must be programmed to ALT0 to be multiplexed as I2C signals.

	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	<reserved>			
GPIO1	High	SCL0	SA4	<reserved>			
GPIO2	High	SDA1	SA3	<reserved>			
GPIO3	High	SCL1	SA2	<reserved>			

LAB 6.1 code description of the "I2C I/O expander device" module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/module.h>  
#include <linux/miscdevice.h>  
#include <linux/i2c.h>  
#include <linux/fs.h>  
#include <linux/of.h>  
#include <linux/uaccess.h>
```
2. You will create a private structure to store the device-specific information. In this driver, the first field of the `ioexp_dev` private structure is an `i2c_client` structure used to handle the I2C device. The second field of the private structure is a `miscdevice` structure. The `misc` subsystem will automatically handle the `open()` function for you; inside the automatically created `open()` function, it will tie your `miscdevice` structure to the `ioexp_dev` one, for the file that's been opened. In this way, in your `write/read` kernel callback functions, you can recover the `miscdevice` structure, which will allow you to get access to the `i2c_client` structure (included in the `ioexp_dev` structure). Once you get the `i2c_client` structure, you

can read/write each I2C specific device by using the SMBus functions. The last field of the private structure is a char array that will store the name of the I2C device.

```
struct ioexp_dev {
    struct i2c_client * client;
    struct miscdevice ioexp_miscdevice;
    char name[8]; /* ioexpXX */
};
```

3. Create a file_operations structure to define which driver functions are called when the user reads and writes to the character devices. This structure will be passed to the misc subsystem when you register a device to it.

```
static const struct file_operations ioexp_fops = {
    .owner = THIS_MODULE,
    .read = ioexp_read_file,
    .write = ioexp_write_file,
};
```

4. In the probe() function, allocate the private structure by calling the devm_kzalloc() function. Initialize each misc device and register it with the kernel by using the misc_register() function. The i2c_set_clientdata() function attaches each allocated private structure to the i2c_client one, which will allow you to access your private data structure in other functions of the driver. For example, you will recover the private structure in each remove() function call (once per each device attached to the bus) by using the i2c_get_clientdata() function.

```
static int ioexp_probe(struct i2c_client * client, const struct i2c_device_id * id)
{
    static int counter = 0;
    struct ioexp_dev * ioexp;

    /* Allocate the private structure */
    ioexp = devm_kzalloc(&client->dev, sizeof(struct ioexp_dev), GFP_KERNEL);

    /* Store pointer to the device-structure in bus device context */
    i2c_set_clientdata(client, ioexp);

    /* Store pointer to I2C client */
    ioexp->client = client;

    /*
     * Initialize the misc device, ioexp is incremented
     * after each probe call
     */
    sprintf(ioexp->name, "ioexp%02d", counter++);
    ioexp->ioexp_miscdevice.name = ioexp->name;
    ioexp->ioexp_miscdevice.minor = MISC_DYNAMIC_MINOR;
    ioexp->ioexp_miscdevice.fops = &ioexp_fops;

    /* Register the misc device */
    return misc_register(&ioexp->ioexp_miscdevice);
}
```

```

        return 0;
    }

```

5. Create the `ioexp_write_file()` kernel callback function, which gets called whenever a user space write operation occurs on one of the character devices. At the time you registered each misc device, you didn't keep any pointer to the private `ioexp_dev` structure. However, as the `miscdevice` structure is accessible through `file->private_data` and is a member of the `ioexp_dev` structure, you can use the `container_of()` macro to compute the address of your private structure and recover the `i2c_client` structure from it. The `copy_from_user()` function will get a char array from user space with values ranging from "0" to "255". This value will be converted from a char string to an unsigned long value, then you will write it to the I2C ioexp device by using the `i2c_smbus_write_byte()` SMBus function. You will also write an `ioexp_read_file()` kernel callback function, which reads the ioexp device input and sends the value to user space. See below a code snippet of the `ioexp_write_file()` function:

```

static ssize_t ioexp_write_file(struct file *file, const char __user *userbuf,
                               size_t count, loff_t *ppos)
{
    int ret;
    unsigned long val;
    char buf[4];
    struct ioexp_dev *ioexp;

    ioexp = container_of(file->private_data, struct ioexp_dev, ioexp_miscdevice);

    copy_from_user(buf, userbuf, count);

    /* Convert char array to char string */
    buf[count-1] = '\0';

    /* Convert the string to an unsigned long */
    ret = kstrtoul(buf, 0, &val);
    i2c_smbus_write_byte(ioexp->client, val);

    return count;
}

```

6. Declare a list of devices supported by the driver:

```

static const struct of_device_id ioexp_dt_ids[] = {
    { .compatible = "arrow,ioexp", },
    { }
};

MODULE_DEVICE_TABLE(of, ioexp_dt_ids);

```

7. Define an array of `i2c_device_id` structures:

```
static const struct i2c_device_id i2c_ids[] = {
    { .name = "ioexp", },
    { }
};
MODULE_DEVICE_TABLE(i2c, i2c_ids);
```

8. Add an `i2c_driver` structure that will be registered to the I2C bus:

```
static struct i2c_driver ioexp_driver = {
    .driver = {
        .name = "ioexp",
        .owner = THIS_MODULE,
        .of_match_table = ioexp_dt_ids,
    },
    .probe = ioexp_probe,
    .remove = ioexp_remove,
    .id_table = i2c_ids,
};
```

9. Register your driver with the I2C bus:

```
module_i2c_driver(ioexp_driver);
```

10. Create a new `io_rpi3_expander.c` file in the `linux_5.4_rpi3_drivers` folder, and add `io_rpi3_expander.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

11. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

12. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 6-1: io_rpi3_expander.c

```
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/i2c.h>
#include <linux/fs.h>
#include <linux/of.h>
#include <linux/uaccess.h>

/* Private device structure */
struct ioexp_dev {
    struct i2c_client *client;
    struct miscdevice ioexp_miscdevice;
    char name[8]; /* ioexpXX */
};

/* User is reading data from /dev/ioexpXX */
static ssize_t ioexp_read_file(struct file *file, char __user *userbuf,
                              size_t count, loff_t *ppos)
{
    int expval, size;
    char buf[3];
    struct ioexp_dev *ioexp;

    ioexp = container_of(file->private_data, struct ioexp_dev, ioexp_miscdevice);

    /* Store IO expander input in expval variable */
    expval = i2c_smbus_read_byte(ioexp->client);
    if (expval < 0)
        return -EFAULT;

    /*
     * Convert expval int value into a char string.
     * For example, 255 int (4 bytes) = FF (2 bytes) + '\0' (1 byte) string.
     */
    size = sprintf(buf, "%02x", expval); /* size is 2 */

    /*
     * Replace NULL by \n. It is not needed to have a char array
     * ended with \0 character.
     */
    buf[size] = '\n';

    /* Send size+1 to include the \n character */
    if(*ppos == 0) {
        if(copy_to_user(userbuf, buf, size+1)) {
            pr_info("Failed to return led_value to user space\n");
            return -EFAULT;
        }
        *ppos+=1;
        return size+1;
    }
}
```

```

    return 0;
}

/* Write from the terminal command line to /dev/ioexpXX, \n is added */
static ssize_t ioexp_write_file(struct file *file, const char __user *userbuf,
                               size_t count, loff_t *ppos)
{
    int ret;
    unsigned long val;
    char buf[4];
    struct ioexp_dev * ioexp;

    ioexp = container_of(file->private_data, struct ioexp_dev, ioexp_miscdevice);

    dev_info(&ioexp->client->dev,
             "ioexp_write_file entered on %s\n", ioexp->name);

    dev_info(&ioexp->client->dev,
             "we have written %zu characters\n", count);

    if(copy_from_user(buf, userbuf, count)) {
        dev_err(&ioexp->client->dev, "Bad copied value\n");
        return -EFAULT;
    }

    buf[count-1] = '\0'; /* replace \n with \0 */

    /* Convert the string to an unsigned long */
    ret = kstrtoul(buf, 0, &val);
    if (ret)
        return -EINVAL;

    dev_info(&ioexp->client->dev, "the value is %lu\n", val);

    ret = i2c_smbus_write_byte(ioexp->client, val);
    if (ret < 0)
        dev_err(&ioexp->client->dev, "the device is not found\n");

    dev_info(&ioexp->client->dev,
             "ioexp_write_file exited on %s\n", ioexp->name);

    return count;
}

static const struct file_operations ioexp_fops = {
    .owner = THIS_MODULE,
    .read = ioexp_read_file,
    .write = ioexp_write_file,
};

/* The probe() function is called after binding */
static int ioexp_probe(struct i2c_client * client,
                      const struct i2c_device_id * id)
{

```



```

static int counter = 0;

struct ioexp_dev * ioexp;

/* Allocate a private structure */
ioexp = devm_kzalloc(&client->dev, sizeof(struct ioexp_dev), GFP_KERNEL);

/* Store pointer to the device-structure in bus device context */
i2c_set_clientdata(client,ioexp);

/* Store pointer to the I2C client device in the private structure */
ioexp->client = client;

/* Initialize the misc device, ioexp is incremented after each probe call */
sprintf(ioexp->name, "ioexp%02d", counter++);
dev_info(&client->dev, "ioexp_probe is entered on %s\n", ioexp->name);

ioexp->ioexp_miscdevice.name = ioexp->name;
ioexp->ioexp_miscdevice.minor = MISC_DYNAMIC_MINOR;
ioexp->ioexp_miscdevice.fops = &ioexp_fops;

/* Register misc device */
return misc_register(&ioexp->ioexp_miscdevice);

dev_info(&client->dev,
        "ioexp_probe is exited on %s\n", ioexp->name);

return 0;
}

static int ioexp_remove(struct i2c_client * client)
{
    struct ioexp_dev * ioexp;

    /* Get device structure from bus device context */
    ioexp = i2c_get_clientdata(client);

    dev_info(&client->dev,
            "ioexp_remove is entered on %s\n", ioexp->name);

    /* Deregister misc device */
    misc_deregister(&ioexp->ioexp_miscdevice);

    dev_info(&client->dev,
            "ioexp_remove is exited on %s\n", ioexp->name);

    return 0;
}

static const struct of_device_id ioexp_dt_ids[] = {
    { .compatible = "arrow,ioexp", },
    { }
};

MODULE_DEVICE_TABLE(of, ioexp_dt_ids);

```

```

static const struct i2c_device_id i2c_ids[] = {
    { .name = "ioexp", },
    { }
};
MODULE_DEVICE_TABLE(i2c, i2c_ids);

static struct i2c_driver ioexp_driver = {
    .driver = {
        .name = "ioexp",
        .owner = THIS_MODULE,
        .of_match_table = ioexp_dt_ids,
    },
    .probe = ioexp_probe,
    .remove = ioexp_remove,
    .id_table = i2c_ids,
};

module_i2c_driver(ioexp_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a driver that controls several I2C IO expanders");

```

io_rpi3_expander.ko demonstration

Scan I2C bus for devices:

```

root@raspberrypi:/home/pi# i2cdetect -y 1
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  39  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --

```

Load the module:

```

root@raspberrypi:/home/pi# insmod io_rpi3_expander.ko
io_rpi3_expander: loading out-of-tree module taints kernel.
ioexp 1-0039: ioexp_probe is entered on ioexp00

```

Check the expander devices connected to the I2C bus:

```

root@raspberrypi:/home/pi# ls -l /dev/ioexp*
crw----- 1 root root 10, 60 Dec  2 22:42 /dev/ioexp00

```

Write to the port (output mode):

```

root@raspberrypi:/home/pi# echo 0 > /dev/ioexp00
ioexp 1-0039: ioexp_write_file entered on ioexp00
ioexp 1-0039: we have written 2 characters

```

```
ioexp 1-0039: the value is 0
ioexp 1-0039: ioexp_write_file exited on ioexp00
```

Read from the port (input mode):

```
root@raspberrypi:/home/pi# cat /dev/ioexp00
00
```

Write to the port (output mode):

```
root@raspberrypi:/home/pi# echo 255 > /dev/ioexp00
ioexp 1-0039: ioexp_write_file entered on ioexp00
ioexp 1-0039: we have written 4 characters
ioexp 1-0039: the value is 255
ioexp 1-0039: ioexp_write_file exited on ioexp00
```

Read from the port (input mode):

```
root@raspberrypi:/home/pi# cat /dev/ioexp00
ff
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod io_rpi3_expander.ko
ioexp 1-0039: ioexp_remove is entered on ioexp00
ioexp 1-0039: ioexp_remove is exited on ioexp00
```

LAB 6.2: "I2C multidisplay LED" module

In this lab, you will implement a driver to control the Analog Devices LTC3206 I2C Multidisplay LED controller (<http://www.analog.com/en/products/power-management/led-driver-ic/inductorless-charge-pump-led-drivers/ltc3206.html>). The LTC3206 provides independent current and dimming control for 1-6 LED MAIN, 1-4 LED SUB and RGB LED Displays with 16 individual dimming states for both the MAIN and SUB displays. Each of the RED, GREEN and BLUE LEDs have 16 dimming states as well, resulting in up to 4096 color combinations. The ENRGB/S (Pin 10) is used to enable and disable either the RED, GREEN and BLUE current sources or the SUB display depending on which is programmed to respond via the I2C port. Once ENRGB/S is brought high, the LTC3206 illuminates the RGB or SUB display with the color combination or intensity that was previously programmed via the I2C port. The logic level of ENRGB/S is referenced to DVCC.

To use the ENRGB/S pin, the I2C port must first be configured to the desired setting. For example, if ENRGB/S will be used to control the SUB display, then a non-zero code must reside in the C3-C0 nibble of the I2C port and bit A2 must be set to 1. Now when ENRGB/S is high (DVCC), the SUB display will be on with the C3-C0 setting. When ENRGB/S is low, the SUB display will be off. If no other displays are programmed to be on, the entire chip will be in shutdown.

Likewise, if ENRGB/S will be used to enable the RGB display, then a non-zero code must reside in one of the RED, GREEN or BLUE nibbles of the serial port (A4-A7 or B0-B7), and bit A2 must be 0. Now when ENRGB/S is high (DVCC), the RGB display will light with the programmed color.

When ENRGB/S is low, the RGB display will be off. If no other displays are programmed to be on, the entire chip will be in shutdown.

If bit A2 is set to 1 (SUB display control), then ENRGB/S will have no effect on the RGB display. Likewise, if bit A2 is set to 0 (RGB display control), then ENRGB/S will have no effect on the SUB display.

If the ENRGB/S pin is not used, it should be connected to DVCC. It should not be grounded or left floating.

In the page 9 of the LTC3206 datasheet you can see the bit assignments.

To test the driver, you can use the DC749A - Demo Board (<http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc749a.html>). You will use the pin 6 of the DC749A J1 connector to control the ENRGB/S pin, connecting it to a GPIO pin of the Raspberry Pi. Connect the SDA signal of the Raspberry Pi to the pin 7 of the J1 connector and the SCL signal of the Raspberry Pi to the pin 4 of the J1 connector. Connect 3.3V between the Raspberry Pi and the J20 DVCC pin. Do not forget to connect GND between DC749A and the Raspberry Pi. If you do not want to enable the ENRGB/S pin, connect it to DVCC.

To develop the driver, you will use the LED subsystem. Every LED Display device (R, G, B, SUB and MAIN) will be registered to the LED subsystem by using the `devm_led_classdev_register()` function, being created five devices (red, green, blue, sub and main) under the `/sys/class/leds/` directory. These five devices will be accessed within the kernel driver space throughout the same driver's `led_control()` function, which will be called every time you write to the brightness sysfs entry from user space. Two additional sysfs entries will be created, allowing to switch from RGB to SUB device and vice versa. These two sysfs functions set the control A2 bit, enabling the ENRGB/S pin to perform the switching.

LAB 6.2 hardware description

You will use the Raspberry Pi GPIO expansion connector to obtain the I2C signals. The GPIO2 and GPIO3 pins will be used to get the SDA1 and SCL1 signals. Connect them to the pin 4 (SCL) and to the pin 7 (SDA) of the DC749A J1 connector. Connect the GPIO23 pin of the Raspberry Pi to the pin 6 (ENRGB/S) of the DC749A J1 connector. Connect 3.3V from the Raspberry Pi board to the Vin J2 pin and J20 DVCC pin of the DC749A J1 connector. Do not forget to connect GND between the two boards.

LAB 6.2 Device Tree description

In the `bcm2710-rpi-3-b.dts` Device Tree file, you can see that the GPIO2 and GPIO3 pins are multiplexed as I2C signals by using the ALT0 mode. You will connect the GPIO23 pin to the pin 6 (ENRGB/S) of the DC749A J1 connector, so the GPIO23 pin must be multiplexed as a GPIO signal.

Open and modify the `bcm2710-rpi-3-b.dts` Device Tree file by adding the `ltc3206@1b` sub-node in the `i2c1` controller master node. The `pinctrl-0` property of the `ltc3206` node points to the `cs_pins` pin configuration node, where the `GPIO23` pin is multiplexed as a GPIO signal. The `gpios` property will make the `GPIO23` available to the driver so that you can set up the pin direction to output and drive the physical line level from 0 to 1 to control the `ENRGB/S` pin. The `reg` property provides the I2C address of the `LTC3206` device. In the `ltc3206` node, there are five sub-nodes representing the different display devices. Each of the five nodes has a `label` property so that the driver can identify and create devices with the label names:

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
    status = "okay";
    [...]

    ltc3206: ltc3206@1b {
        compatible = "arrow,ltc3206";
        reg = <0x1b>;
        pinctrl-0 = <&cs_pins>;
        gpios = <&gpio 23 GPIO_ACTIVE_LOW>;

        led1r {
            label = "red";
        };

        led1b {
            label = "blue";
        };

        led1g {
            label = "green";
        };

        ledmain {
            label = "main";
        };

        ledsub {
            label = "sub";
        };
    };
};
```

See below the `cs_pins` pin configuration node, where the `GPIO23` pin is multiplexed as a GPIO signal:

```
cs_pins: cs_pins {
    brcm,pins = <23>;
```

```

        brcm,function = <1>;      /* Output */
        brcm,pull = <0>;          /* none */

};

```

Unified device properties interface for ACPI and Device Tree

At boot, the Linux kernel needs to retrieve hardware information about the system for which it has been compiled. This hardware information is stored using two main methods: Device Tree and ACPI tables. A Unified Device Properties API has been defined to provide a format compatible with both existing methods of storing the hardware data. The purpose of this is the development of firmware agnostic device drivers. In order to support this, it is needed to convert the Linux driver to use unified device property functions instead of DT specific.

The following functions in bold are provided in analogy with the corresponding functions for the device structure:

```

fwnode_property_present() for device_property_present()
fwnode_property_read_u8() for device_property_read_u8()
fwnode_property_read_u16() for device_property_read_u16()
fwnode_property_read_u32() for device_property_read_u32()
fwnode_property_read_u64() for device_property_read_u64()
fwnode_property_read_string() for device_property_read_string()
fwnode_property_read_u8_array() for device_property_read_u8_array()
fwnode_property_read_u16_array() for device_property_read_u16_array()
fwnode_property_read_u32_array() for device_property_read_u32_array()
fwnode_property_read_u64_array() for device_property_read_u64_array()
fwnode_property_read_string_array() for device_property_read_string_array()

```

For all of these functions, the first argument is a pointer to a `fwnode_handle` structure (declared in `include/linux/fwnode.h` in the kernel source tree), which allows a device description object (depending on what platform firmware interface is in use) to be obtained.

```

/* fwnode.h - Firmware device node object handle type definition. */

enum fwnode_type {
    FWNODE_INVALID = 0,
    FWNODE_OF,
    FWNODE_ACPI,
    FWNODE_ACPI_DATA,
    FWNODE_PDATA,
    FWNODE_IRQCHIP,
};

struct fwnode_handle {
    enum fwnode_type type;
    struct fwnode_handle *secondary;
};

```

The function `device_for_each_child_node()` iterates over the children of the device description object associated with a given device (for example, `struct device *dev = &client->dev`), and the function `device_get_child_node_count()` returns the number of child nodes of a given device.

LAB 6.2 code description of the "I2C multidisplay LED" module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/leds.h>
#include <linux/gpio/consumer.h>
#include <linux/delay.h>
```

2. Define the masks that will be used to select the specific I2C commands:

```
#define CMD_RED_SHIFT      4
#define CMD_BLUE_SHIFT     4
#define CMD_GREEN_SHIFT   0
#define CMD_MAIN_SHIFT     4
#define CMD_SUB_SHIFT      0
#define EN_CS_SHIFT       (1 << 2)
```

3. Create a private structure named `led_device` that will store specific data for each of the five LED devices. The first field of the `led_device` structure is the brightness variable, that will hold values ranging from "0" to "15". The second field is an `led_classdev` structure, that will be filled for each led device within the `probe()` function. The last field is a pointer to a private structure that will hold global data shared with all the LED devices. This global structure will be analyzed in the next point.

```
struct led_device {
    u8 brightness;
    struct led_classdev cdev;
    struct led_priv *private;
};
```

4. Create a private structure that will store global data accessible to all the LED devices. The first field of the private structure is the `num_leds` variable, which will hold the number of led devices declared in the Device Tree. The second field is an array of three commands that will hold the command values sent to the LTC3206 device in each of the I2C transactions. The `display_cs` variable is a pointer to a `gpio_desc` structure that will allow you to control the ENRGB/S pin, and the last field is a pointer to an `i2c_client` structure that will allow you to recover the I2C address of the LTC3206 device.

```
struct led_priv {
    u32 num_leds;
    u8 command[3];
};
```

```

    struct gpio_desc *display_cs;
    struct i2c_client *client;
};

```

5. These are the main points to set up the driver within the `probe()` function:

- Declare a pointer to a `fwnode_handle` structure and a pointer to the `led_priv` global structure.
- Get the number of LED devices by calling the `device_get_child_node_count()` function.
- Allocate the global structure by calling `devm_kzalloc()`, and store the pointer to your client device on it (`private->client = client`). The `i2c_set_clientdata()` function attaches your private structure to the `i2c_client` one.
- Get the gpio descriptor, and store it in the global private structure (`private->display_cs = devm_gpiod_get(dev, NULL, GPIOD_ASIS)`). Set the gpio pin direction to output and the pin physical level to low (`gpiod_direction_output(private->display_cs, 1)`). In one of the gpios property fields of the Device Tree, is declared `GPIO_ACTIVE_LOW`, meaning that `gpiod_set_value(desc, 1)` will set the physical line to low and `gpiod_set_value(desc, 0)` to high.
- The `device_for_each_child_of_node()` function walks through each LED child node, allocating an `led_device` private structure for each one by using the `devm_kzalloc()` function and initializing the `led_classdev` structure included in each private structure. The `fwnode_property_read_string()` function reads each LED node's label property and stores it in the `cdev->name` field of each `led_device` structure.
- The `devm_led_classdev_register()` function registers each LED class device to the LED subsystem.
- Finally, add a group of "sysfs attribute files" to control the ENRGB/S pin by using the function `sysfs_create_group()`.

```

static int __init ltc3206_probe(struct i2c_client *client,
                               const struct i2c_device_id *id)
{
    struct fwnode_handle *child;
    struct device *dev = &client->dev;
    struct led_priv *private;

    device_get_child_node_count(dev);

    private = devm_kzalloc(dev, sizeof(*private), GFP_KERNEL);
    private->client = client;
    i2c_set_clientdata(client, private);

    private->display_cs = devm_gpiod_get(dev, NULL, GPIOD_ASIS);
    gpiod_direction_output(private->display_cs, 1);

    /* Register sysfs hooks */
    sysfs_create_group(&client->dev.kobj, &display_cs_group);
}

```



```

/* Do an iteration for each child node */
device_for_each_child_node(dev, child) {

    struct led_device *led_device;
    struct led_classdev *cdev;

    led_device = devm_kzalloc(dev, sizeof(*led_device), GFP_KERNEL);

    cdev = &led_device->cdev;
    led_device->private = private;

    fwnode_property_read_string(child, "label", &cdev->name);

    if (strcmp(cdev->name,"main") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        devm_led_classdev_register(dev, &led_device->cdev);
    }
    else if (strcmp(cdev->name,"sub") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        devm_led_classdev_register(dev, &led_device->cdev);
    }
    else if (strcmp(cdev->name,"red") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        ret = devm_led_classdev_register(dev, &led_device->cdev);
    }
    else if (strcmp(cdev->name,"green") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        ret = devm_led_classdev_register(dev, &led_device->cdev);
    }
    else if (strcmp(cdev->name,"blue") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        ret = devm_led_classdev_register(dev, &led_device->cdev);
    }
    else {
        dev_err(dev, "Bad device tree value\n");
        return -EINVAL;
    }

    private->num_leds++;

}

dev_info(dev, "i am out of the device tree\n");
dev_info(dev, "my_probe() function is exited.\n");
return 0;
}

```

6. Write the LED brightness `led_control()` function. Every time your user space application writes to the brightness sysfs entry (`/sys/class/leds/<device>/brightness`) under each LED device, the driver's `led_control()` function is called. The LED subsystem hides the complexity of creating a class, the devices under the class and the sysfs entries under each of the devices. The `led_device` structure associated with each device is recovered by using the `container_of()` function. Depending of the `cdev->name` value (included in the

led_device structure), different masks are applied to the char array command values, then the updated values are stored in the led_priv global structure. Finally, you will send the updated command values to the LTC3206 device by using the ltc3206_led_write() function, which calls to the plain i2c_master_send() function.

7. In the probe() function, you added a group of "sysfs attribute files" to control the ENRGB/S pin by writing the line of code sysfs_create_group(&client->dev.kobj, &display_cs_group). Now, you will create two structures of type device_attribute with the respective names 'rgb' and 'sub', and you will organize these two attributes into a group:

```
static DEVICE_ATTR(rgb, S_IWUSR, NULL, rgb_select);
static DEVICE_ATTR(sub, S_IWUSR, NULL, sub_select);

static struct attribute *display_cs_attrs[] = {
    &dev_attr_rgb.attr,
    &dev_attr_sub.attr,
    NULL,
};

static struct attribute_group display_cs_group = {
    .name = "display_cs",
    .attrs = display_cs_attrs,
};
```

8. Write the rgb_select() and sub_select() functions that will be called each time the user application writes "on" or "off" to the rgb" and "sub" sysfs entries. Inside these functions, you will recover the i2c_client structure by using the to_i2c_client() function, then the i2c_get_clientdata() function will recover the led_priv global structure. The i2c_get_clientdata() function takes the previously recovered i2c_client structure as a parameter. Once you have retrieved the global structure, you can update the bit A2 of the command[0] by using the mask EN_CS_SHIFT, then you will send the new command values to the LTC3206 device by using the ltc3206_led_write() function. Depending of the selected "on" or "off" value, the GPIO physical line will be set from "low to high" or from "high to low" by using the gpiod_set_value() function, which takes as a parameter the gpio descriptor stored in your led_priv global structure.
9. Declare a list of devices supported by the driver:

```
static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow, ltc3206", },
    { }
};
MODULE_DEVICE_TABLE(of, my_of_ids);
```

10. Define an array of i2c_device_id structures:

```
static const struct i2c_device_id ltc3206_id[] = {
    { "ltc3206", 0 },
};
```

```

    { }
};
MODULE_DEVICE_TABLE(i2c, ltc3206_id);

```

11. Add an `i2c_driver` structure that will be registered to the I2C bus:

```

static struct i2c_driver ltc3206_driver = {
    .probe =      ltc3206_probe,
    .remove =     ltc3206_remove,
    .id_table =   ltc3206_id,
    .driver = {
        .name = "ltc3206",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};

```

12. Register your driver with the I2C bus:

```

module_i2c_driver(ltc3206_driver);

```

13. Create a new `ltc3206_rpi3_led_class.c` file in the `linux_5.4_rpi3_drivers` folder, and add `ltc3206_rpi3_led_class.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```

~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy

```

14. Build the modified Device Tree, and load it to the target processor:

```

~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/

```

15. Reboot the Raspberry Pi:

```

root@raspberrypi:/home/pi# reboot

```

Listing 6-2: ltc3206_rpi3_led_class.c

```

#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/leds.h>
#include <linux/gpio/consumer.h>
#include <linux/delay.h>

#define LED_NAME_LEN      32
#define CMD_RED_SHIFT     4
#define CMD_BLUE_SHIFT    4
#define CMD_GREEN_SHIFT   0
#define CMD_MAIN_SHIFT    4
#define CMD_SUB_SHIFT     0
#define EN_CS_SHIFT       (1 << 2)

/* Set an led_device struct for each of the 5 led devices */
struct led_device {
    u8 brightness;
    struct led_classdev cdev;
    struct led_priv *private;
};

/*
 * Store the global parameters shared for the 5 led devices.
 * The parameters are updated after each led_control() call
 */
struct led_priv {
    u32 num_leds;
    u8 command[3];
    struct gpio_desc *display_cs;
    struct i2c_client *client;
};

/* Function that writes to the I2C device */
static int ltc3206_led_write(struct i2c_client *client, u8 *command)
{
    int ret = i2c_master_send(client, command, 3);
    if (ret >= 0)
        return 0;
    return ret;
}

/* The sysfs functions */
static ssize_t sub_select(struct device *dev, struct device_attribute *attr,
                          const char *buf, size_t count)
{
    char *buffer;
    struct i2c_client *client;
    struct led_priv *private;

    buffer = buf;

```

```

/* Replace \n added from terminal with \0 */
*(buffer+(count-1)) = '\0';

client = to_i2c_client(dev);
private = i2c_get_clientdata(client);

private->command[0] |= EN_CS_SHIFT; /* set the 3d bit A2 */
ltc3206_led_write(private->client, private->command);

if(!strcmp(buffer, "on")) {
    gpiod_set_value(private->display_cs, 1); /* low */
    usleep_range(100, 200);
    gpiod_set_value(private->display_cs, 0); /* high */
}
else if (!strcmp(buffer, "off")) {
    gpiod_set_value(private->display_cs, 0); /* high */
    usleep_range(100, 200);
    gpiod_set_value(private->display_cs, 1); /* low */
}
else {
    dev_err(&client->dev, "Bad led value.\n");
    return -EINVAL;
}

return count;
}

static DEVICE_ATTR(sub, S_IWUSR, NULL, sub_select);

static ssize_t rgb_select(struct device *dev, struct device_attribute *attr,
                          const char *buf, size_t count)
{
    char *buffer;
    struct i2c_client *client = to_i2c_client(dev);
    struct i2c_priv *private = i2c_get_clientdata(client);
    buffer = buf;

    *(buffer+(count-1)) = '\0';

    private->command[0] &= ~(EN_CS_SHIFT); /* clear the 3d bit */

    ltc3206_led_write(private->client, private->command);

    if(!strcmp(buffer, "on")) {
        gpiod_set_value(private->display_cs, 1); /* low */
        usleep_range(100, 200);
        gpiod_set_value(private->display_cs, 0); /* high */
    }
    else if (!strcmp(buffer, "off")) {
        gpiod_set_value(private->display_cs, 0); /* high */
        usleep_range(100, 200);
        gpiod_set_value(private->display_cs, 1); /* low */
    }
    else {
        dev_err(&client->dev, "Bad led value.\n");
    }
}

```

```

        return -EINVAL;
    }

    return count;
}
static DEVICE_ATTR(rgb, S_IWUSR, NULL, rgb_select);

static struct attribute *display_cs_attrs[] = {
    &dev_attr_rgb.attr,
    &dev_attr_sub.attr,
    NULL,
};

static struct attribute_group display_cs_group = {
    .name = "display_cs",
    .attrs = display_cs_attrs,
};

/*
 * This is the function that is called
 * when you write the brightness file under each device.
 * The command parameters are stored in the led_priv structure
 * that is pointed inside each led_device structure
 */
static int led_control(struct led_classdev *led_cdev, enum led_brightness value)
{
    struct led_classdev *cdev;
    struct led_device *led;
    led = container_of(led_cdev, struct led_device, cdev);
    cdev = &led->cdev;
    led->brightness = value;

    dev_info(cdev->dev, "the subsystem is %s\n", cdev->name);

    if (value > 15 || value < 0)
        return -EINVAL;

    if (strcmp(cdev->name, "red") == 0) {
        led->private->command[0] &= 0x0F; /* clear the upper nibble */
        led->private->command[0] |= ((led->brightness << CMD_RED_SHIFT) & 0xF0);
    }
    else if (strcmp(cdev->name, "blue") == 0) {
        led->private->command[1] &= 0x0F; /* clear the upper nibble */
        led->private->command[1] |= ((led->brightness << CMD_BLUE_SHIFT) & 0xF0);
    }
    else if (strcmp(cdev->name, "green") == 0) {
        led->private->command[1] &= 0xF0; /* clear the lower nibble */
        led->private->command[1] |= ((led->brightness << CMD_GREEN_SHIFT) & 0x0F);
    }
    else if (strcmp(cdev->name, "main") == 0) {
        led->private->command[2] &= 0x0F; /* clear the upper nibble */
        led->private->command[2] |= ((led->brightness << CMD_MAIN_SHIFT) & 0xF0);
    }
    else if (strcmp(cdev->name, "sub") == 0) {

```

```

        led->private->command[2] &= 0xF0; /* clear the lower nibble */
        led->private->command[2] |= ((led->brightness << CMD_SUB_SHIFT) & 0x0F);
    }
    else
        dev_info(cdev->dev, "No display found\n");

    return ltc3206_led_write(led->private->client, led->private->command);
}

static int __init ltc3206_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    int count, ret;
    u8 value[3];
    struct fwnode_handle *child;
    struct device *dev = &client->dev;
    struct led_priv *private;

    dev_info(dev, "platform_probe enter\n");

    /*
     * Set blue LED to maximum value for I2C testing.
     * ENRGB must be set to VCC to do the testing
     */
    value[0] = 0x00;
    value[1] = 0xF0;
    value[2] = 0x00;

    i2c_master_send(client, value, 3);

    dev_info(dev, "led BLUE is ON\n");

    count = device_get_child_node_count(dev);
    if (!count)
        return -ENODEV;

    dev_info(dev, "there are %d nodes\n", count);

    private = devm_kzalloc(dev, sizeof(*private), GFP_KERNEL);
    if (!private)
        return -ENOMEM;

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

    private->display_cs = devm_gpiod_get(dev, NULL, GPIOD_ASIS);
    if (IS_ERR(private->display_cs)) {
        ret = PTR_ERR(private->display_cs);
        dev_err(dev, "Unable to claim gpio\n");
        return ret;
    }

    gpiod_direction_output(private->display_cs, 1);

    /* Register sysfs hooks */

```

```

ret = sysfs_create_group(&client->dev.kobj, &display_cs_group);
if (ret < 0) {
    dev_err(&client->dev, "couldn't register sysfs group\n");
    return ret;
}

/* Parse all the child nodes */
device_for_each_child_node(dev, child) {

    struct led_device *led_device;
    struct led_classdev *cdev;

    led_device = devm_kzalloc(dev, sizeof(*led_device), GFP_KERNEL);
    if (!led_device)
        return -ENOMEM;

    cdev = &led_device->cdev;
    led_device->private = private;

    fwnode_property_read_string(child, "label", &cdev->name);

    if (strcmp(cdev->name, "main") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        ret = devm_led_classdev_register(dev, &led_device->cdev);
        if (ret)
            goto err;
        dev_info(cdev->dev, "the subsystem is %s and num is %d\n",
                 cdev->name, private->num_leds);
    }
    else if (strcmp(cdev->name, "sub") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        ret = devm_led_classdev_register(dev, &led_device->cdev);
        if (ret)
            goto err;
        dev_info(cdev->dev, "the subsystem is %s and num is %d\n",
                 cdev->name, private->num_leds);
    }
    else if (strcmp(cdev->name, "red") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        ret = devm_led_classdev_register(dev, &led_device->cdev);
        if (ret)
            goto err;
        dev_info(cdev->dev, "the subsystem is %s and num is %d\n",
                 cdev->name, private->num_leds);
    }
    else if (strcmp(cdev->name, "green") == 0) {
        led_device->cdev.brightness_set_blocking = led_control;
        ret = devm_led_classdev_register(dev, &led_device->cdev);
        if (ret)
            goto err;
        dev_info(cdev->dev, "the subsystem is %s and num is %d\n",
                 cdev->name, private->num_leds);
    }
    else if (strcmp(cdev->name, "blue") == 0) {

```



```

        led_device->cdev.brightness_set_blocking = led_control;
        ret = devm_led_classdev_register(dev, &led_device->cdev);
        if (ret)
            goto err;
        dev_info(cdev->dev, "the subsystem is %s and num is %d\n",
                cdev->name, private->num_leds);
    }
    else {
        dev_err(dev, "Bad device tree value\n");
        return -EINVAL;
    }

    private->num_leds++;
}

dev_info(dev, "i am out of the device tree\n");
dev_info(dev, "my_probe() function is exited.\n");
return 0;

err:
    fwnode_handle_put(child);
    sysfs_remove_group(&client->dev.kobj, &display_cs_group);
    return ret;
}

static int ltc3206_remove(struct i2c_client *client)
{
    dev_info(&client->dev, "leds_remove enter\n");
    sysfs_remove_group(&client->dev.kobj, &display_cs_group);
    dev_info(&client->dev, "leds_remove exit\n");

    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,ltc3206"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static const struct i2c_device_id ltc3206_id[] = {
    { "ltc3206", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, ltc3206_id);

static struct i2c_driver ltc3206_driver = {
    .probe =      ltc3206_probe,
    .remove =     ltc3206_remove,
    .id_table =   ltc3206_id,
    .driver = {
        .name = "ltc3206",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    },
};

```

```

    }
};
module_i2c_driver(ltc3206_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a driver that controls the ltc3206 I2C multidisplay device");

```

ltc3206_rpi3_led_class.ko demonstration

Scan I2C bus for devices:

```

root@raspberrypi:/home/pi# i2cdetect -y 1
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  1b  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --

```

In the LTC3206 board, connect the ENRGB/S pin to DVCC.

Load the module. The probe() function is called, and the blue LED is ON:

```

root@raspberrypi:/home/pi# insmod ltc3206_rpi3_led_class.ko
ltc3206 1-001b: platform_probe enter
ltc3206 1-001b: led BLUE is ON
ltc3206 1-001b: there are 5 nodes
leds red: the subsystem is red and num is 0
leds blue: the subsystem is blue and num is 1
leds green: the subsystem is green and num is 2
leds main: the subsystem is main and num is 3
leds sub: the subsystem is sub and num is 4
ltc3206 1-001b: i am out of the device tree
ltc3206 1-001b: my_probe() function is exited.

```

Check all the devices under the leds class:

```

root@raspberrypi:/home/pi# ls -l /sys/class/leds
total 0
lrwxrwxrwx 1 root root 0 Apr  8 18:43 blue -> ../../devices/platform/soc/3f80400
0.i2c/i2c-1/1-001b/leds/blue
lrwxrwxrwx 1 root root 0 Apr  8 18:43 default-on -> ../../devices/virtual/leds/d
efault-on
lrwxrwxrwx 1 root root 0 Apr  8 18:43 green -> ../../devices/platform/soc/3f80400
0.i2c/i2c-1/1-001b/leds/green
lrwxrwxrwx 1 root root 0 Apr  8 18:43 led0 -> ../../devices/platform/leds/leds/l
ed0
lrwxrwxrwx 1 root root 0 Apr  8 18:43 led1 -> ../../devices/platform/leds/leds/l
ed1
lrwxrwxrwx 1 root root 0 Apr  8 18:43 main -> ../../devices/platform/soc/3f80400

```

```
0.i2c/i2c-1/1-001b/leds/main
lrwxrwxrwx 1 root root 0 Apr  8 18:43 mmc0 -> ../../devices/virtual/leds/mmc0
lrwxrwxrwx 1 root root 0 Apr  8 18:43 red -> ../../devices/platform/soc/3f804000
.i2c/i2c-1/1-001b/leds/red
lrwxrwxrwx 1 root root 0 Apr  8 18:43 sub -> ../../devices/platform/soc/3f804000
.i2c/i2c-1/1-001b/leds/sub
```

Change the brightness of the RGB, MAIN and SUB displays of the LTC3206 device. The maximum brightness value is 15 and the minimum is 0:

```
root@raspberrypi:/home/pi# echo 10 > /sys/class/leds/red/brightness
root@raspberrypi:/home/pi# echo 15 > /sys/class/leds/red/brightness
root@raspberrypi:/home/pi# echo 0 > /sys/class/leds/red/brightness
root@raspberrypi:/home/pi# echo 10 > /sys/class/leds/blue/brightness
root@raspberrypi:/home/pi# echo 15 > /sys/class/leds/blue/brightness
root@raspberrypi:/home/pi# echo 0 > /sys/class/leds/blue/brightness
root@raspberrypi:/home/pi# echo 10 > /sys/class/leds/green/brightness
root@raspberrypi:/home/pi# echo 15 > /sys/class/leds/green/brightness
root@raspberrypi:/home/pi# echo 0 > /sys/class/leds/green/brightness
root@raspberrypi:/home/pi# echo 10 > /sys/class/leds/main/brightness
root@raspberrypi:/home/pi# echo 15 > /sys/class/leds/main/brightness
root@raspberrypi:/home/pi# echo 0 > /sys/class/leds/main/brightness
root@raspberrypi:/home/pi# echo 10 > /sys/class/leds/sub/brightness
root@raspberrypi:/home/pi# echo 15 > /sys/class/leds/sub/brightness
root@raspberrypi:/home/pi# echo 0 > /sys/class/leds/sub/brightness
```

Mix RED, GREEN, and BLUE colors for the RGB display:

```
root@raspberrypi:/home/pi# echo 15 > /sys/class/leds/red/brightness
root@raspberrypi:/home/pi# echo 15 > /sys/class/leds/blue/brightness
root@raspberrypi:/home/pi# echo 15 > /sys/class/leds/green/brightness
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod ltc3206_rpi3_led_class.ko
```

Switch off the power supply and connect the ENRGB/S pin of the LTC3206 board to the GPIO23 pin of the Raspberry Pi board. Switch on the power supply to boot the Raspberry Pi.

Load the module:

```
root@raspberrypi:/home/pi# insmod ltc3206_rpi3_led_class.ko
```

Switch on the SUB display with brightness = 10. The SUB display is ON:

```
root@raspberrypi:/home/pi# echo 10 > /sys/class/leds/sub/brightness
```

Switch on the red LED with brightness = 10. The red LED is OFF:

```
root@raspberrypi:/home/pi# echo 10 > /sys/class/leds/red/brightness
```

Switch the SUB display OFF, and switch the red LED ON:

```
root@raspberrypi:/home/pi# echo off > /sys/class/i2c-dev/i2c-1/device/1-001b/display_cs/sub
```

Switch the red LED OFF, and switch the SUB display ON:

```
root@raspberrypi:/home/pi# echo off > /sys/class/i2c-dev/i2c-1/device/1-001b/display_cs/rgb
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod ltc3206_rpi3_led_class.ko
```

7

Handling Interrupts in Device Drivers

An IRQ is an interrupt request which can come from different sources, such as a GPIO, EXTI, or on-chip peripherals. Different devices can use the same interrupt line, thus sharing an IRQ.

In Linux, the **IRQ number** is an enumeration of the different interrupt sources on a machine. Typically, what is enumerated is the number of input pins on all of the interrupt controllers in the system. The IRQ number is a **virtual interrupt ID** and hardware independent.

The Linux kernel uses a single large number space where each separate IRQ source is assigned a different number. This is simple when there is only one interrupt controller, but in systems with multiple interrupt controllers, the kernel must ensure that each one gets assigned non-overlapping allocations of Linux IRQ numbers.

In modern SoCs, the number of **interrupt controllers** registered as **irqchips** is growing. Whereas in the past, IRQ numbers could be chosen so that they matched the hardware IRQ line into the root interrupt controller, nowadays this number is just a number. For this reason, we need a mechanism to separate controller-local interrupt numbers, called **hardware irq's (hwirq)**, from Linux IRQ numbers.

An interrupt controller driver, dependent of a specific architecture, registers an **irq_chip** structure to the kernel. This structure contains a group of pointers to the functions that are going to manage the IRQs of the interrupt controller. The struct `irq_chip` is declared as follows:

```
/*
 * struct irq_chip - hardware interrupt chip descriptor
 * @parent_device: pointer to parent device for irqchip
 * @name:          name for /proc/interrupts
 * @irq_startup:   start up the interrupt (defaults to ->enable if NULL)
 * @irq_shutdown:  shut down the interrupt (defaults to ->disable if NULL)
 * @irq_enable:    enable the interrupt (defaults to chip->unmask if NULL)
 * @irq_disable:   disable the interrupt
 * @irq_ack:       start of a new interrupt
 * @irq_mask:      mask an interrupt source
 * @irq_mask_ack:  ack and mask an interrupt source
```

```

* @irq_unmask:           unmask an interrupt source
* @irq_eoi:             end of interrupt
* @irq_set_affinity:     set the CPU affinity on SMP machines
* @irq_retrigger:       resend an IRQ to the CPU
* @irq_set_type:        set the flow type (IRQ_TYPE_LEVEL/etc.) of an IRQ
* @irq_set_wake:        enable/disable power-management wake-on of an IRQ

[...]

*/
struct irq_chip {
    struct device      *parent_device;
    const char        *name;
    unsigned int       (*irq_startup)(struct irq_data *data);
    void              (*irq_shutdown)(struct irq_data *data);
    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_mask_ack)(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);
    void              (*irq_cpu_online)(struct irq_data *data);
    void              (*irq_cpu_offline)(struct irq_data *data);

    [...]

};

```

The `irq_chip` structure contains all the direct chip relevant functions which can be utilized by the IRQ flow implementations. These primitives mean exactly what their name says: `ack` means ACK, `mask` means masking of an IRQ line, etc. It is up to the flow handler(s) to use these basic units of low-level functionality.

A Linux IRQ number is always tied to an `irq_desc` structure, which is the structure that represents an IRQ. A list of IRQ descriptors are maintained in an array (indexed by the IRQ number) called the IRQ descriptor table. The `handle_irq` element of struct `irq_desc` is a function pointer of type `irq_flow_handler_t` which refers to a high-level function that deals with flow management on the line (typedef `void (*irq_flow_handler_t)(struct irq_desc *desc);`). Whenever an interrupt triggers, the low level arch code calls into the generic interrupt code by calling `irq_desc->handle_irq`. This high level

IRQ handling function only uses `irq_desc->irq_data->chip` primitives referenced by the assigned chip descriptor structure. The struct `irq_desc` is declared as follows:

```
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;
#ifdef CONFIG_IRQ_PREFLOW_FASTEOI
    irq_preflow_handler_t     preflow_handler;
#endif
    struct irqaction          *action; /* IRQ action list */
    unsigned int              status_use_accessors;
    unsigned int              core_internal_state__do_not_mess_with_it;
    unsigned int              depth; /* nested irq disables */
    unsigned int              wake_depth; /* nested wake enables */
    unsigned int              irq_count; /* For detecting broken IRQs */
    unsigned long             last_unhandled; /* Aging timer unhandled count */
    unsigned int              irq_unhandled;
    atomic_t                  threads_handled;
    int                       threads_handled_last;
    raw_spinlock_t            lock;
    struct cpumask             *percpu_enabled;
    const struct cpumask      *percpu_affinity;
#ifdef CONFIG_SMP
    const struct cpumask      *affinity_hint;
    struct irq_affinity_notify *affinity_notify;
#endif
#ifdef CONFIG_GENERIC_PENDING_IRQ
    cpumask_var_t             pending_mask;
    [...]
}
```

Inside each `irq_desc` structure, there is an instance of `irq_data` (in bold in `irq_desc` above). The `irq_desc` structure contains low-level information that is relevant for interrupt management, such as Linux IRQ number, hwirq number, interrupt translation domain (`irq_domain`) and a pointer to interrupt controller operations (`irq_chip`) among other important fields.

```
/*
 * struct irq_data - per irq chip data passed down to chip functions
 * @mask:           precomputed bitmask for accessing the chip registers
 * @irq:            interrupt number
 * @hwirq:          hardware interrupt number, local to the interrupt domain
 * @common:         point to data shared by all irqchips
 * @chip:           low level interrupt hardware access
 * @domain:         Interrupt translation domain; responsible for mapping
 *                  between hwirq number and linux irq number.
 * @parent_data:    pointer to parent struct irq_data to support hierarchy
 *                  irq_domain
 * @chip_data:      platform-specific per-chip private data for the chip
 *                  methods, to allow shared chip implementations
 */
```

```

*/
struct irq_data {
    u32                                mask;
    unsigned int                       irq;    /* linux IRQ number */
    unsigned long                      hwirq; /* hwirq number */
    struct irq_common_data             *common;
    struct irq_chip                    *chip; /* low level int controller hw access */
    struct irq_domain                  *domain;
#ifdef CONFIG_IRQ_DOMAIN_HIERARCHY
    struct irq_data                    *parent_data;
#endif
    void                               *chip_data;
};

```

Linux kernel IRQ domain for GPIO controllers

The kernel internals use a single number space to represent IRQ numbers; there are no two IRQs having the same numbers. This is also true when an embedded processor has a single interrupt controller (IC). However, a mapping is needed when the processor has two interrupt controllers available. To solve this problem, the Linux kernel is using the **IRQ domain**, which is a translation interface between hardware IRQ numbers and the one used internally in the kernel.

The `irq_domain` structure is the interrupt controller "domain" data structure. It handles the mapping between hardware and virtual interrupt numbers for a given interrupt domain. The `irq_desc` structure contains a pointer to the `irq_domain` structure (this pointer is included in the `struct irq_data` field of the `irq_desc` structure).

```

struct irq_domain {
    struct list_head link;
    const char *name;
    const struct irq_domain_ops *ops;
    void *host_data;
    unsigned int flags;

    /* Optional data */
    struct fwnode_handle *fwnode;
    enum irq_domain_bus_token bus_token;
    struct irq_domain_chip_generic *gc;
#ifdef CONFIG_IRQ_DOMAIN_HIERARCHY
    struct irq_domain *parent;
#endif

    /* reverse map data. The linear map gets appended to the irq_domain */
    irq_hw_number_t hwirq_max;
    unsigned int revmap_direct_max_irq;
    unsigned int revmap_size;
    struct radix_tree_root revmap_tree;
    unsigned int linear_revmap[];
};

```


An interrupt controller driver allocates and registers an `irq_domain` by calling one of the `irq_domain_add_*`() functions. The function will return a pointer to the `irq_domain` structure on success. The driver must provide to the chosen allocator function an `irq_domain_ops` structure as an argument. There are several mechanisms available for reverse mapping from hwirq to Linux IRQ, and each mechanism uses a different allocation function. The majority of drivers should use the linear map through the `irq_domain_add_linear()` function:

```
/*
 * irq_domain_add_linear() - Allocate and register a linear revmap irq_domain.
 * @of_node: pointer to interrupt controller's device tree node.
 * @size: Number of interrupts in the domain, eg., the number of GPIO inputs
 * @ops: map/unmap domain callbacks
 * @host_data: Controller private data pointer
 */
struct irq_domain *irq_domain_add_linear(struct device_node *of_node,
                                         unsigned int size,
                                         const struct irq_domain_ops *ops,
                                         void *host_data)
{
    return __irq_domain_add(of_node_to_fwnode(of_node), size,
                           size, 0, ops, host_data);
}
```

In most cases, the `irq_domain` will begin empty without any mappings between the hwirq and IRQ numbers. The `irq_domain` structure is filled with the IRQ mapping by calling `irq_create_mapping()`, which accepts the `irq_domain` structure and a hwirq number as arguments and returns the Linux IRQ number:

```
unsigned int irq_create_mapping(struct irq_domain *domain,
                              irq_hw_number_t hwirq)
{
    struct device_node *of_node;
    int virq;

    [...]

    of_node = irq_domain_get_of_node(domain);

    /* Check if mapping already exists */
    virq = irq_find_mapping(domain, hwirq);
    if (virq) {
        pr_debug("-> existing mapping on virq %d\n", virq);
        return virq;
    }

    /* Allocate a Linux IRQ number */
    virq = irq_domain_alloc_descs(-1, 1, hwirq, of_node_to_nid(of_node), NULL);
    if (virq <= 0) {
        pr_debug("-> virq allocation failed\n");
        return 0;
    }
}
```

```

    if (irq_domain_associate(domain, virq, hwirq)) {
        irq_free_desc(virq);
        return 0;
    }

    pr_debug("irq %lu on domain %s mapped to virtual irq %u\n",
            hwirq, of_node_full_name(of_node), virq);

    return virq;
}

```

When writing drivers for GPIO controllers that are also interrupt controllers, `irq_create_mapping()` can be called from within `gpio_chip.to_irq()` callback function. This callback function is called whenever a driver calls the `gpiod_to_irq()` function to get the Linux IRQ number associated with the GPIO pin of a GPIO interrupt controller.

If a mapping for the `hwirq` doesn't already exist, then `irq_create_mapping()` will allocate a new `irq_desc` structure, associate it with the `hwirq` and call the `irq_domain_ops.map()` callback (by means of the `irq_domain_associate()` function) so that the driver can perform any required hardware setup. In the `.map()` function is created a mapping between a Linux IRQ number and a `hwirq` number. The mapping is done inside `.map()` by calling the `irq_set_chip_and_handler()` function. The third parameter (`handle`) of `irq_set_chip_and_handler()` determines the wrapper function that will call the real handler (registered by using `request_irq()` or `request_threaded_irq()`) for the "GPIO device" driver that caused the interrupt.

The GPIO irqchips usually fall into one of two categories:

1. **CHAINED GPIO irqchips:** These are usually the type that is embedded on an SoC. This means that there is a fast IRQ handler for the GPIOs that gets called in a chain from the parent IRQ handler, most typically the system interrupt controller. The GPIO irqchip will register to its parent IRQ handler by calling `irq_set_chained_handler_and_data()`, then the GPIO irqchip handler (which is passed as a parameter to the `irq_set_chained_handler_and_data()` function, along with the parent IRQ of the GPIO irqchip) will be called immediately from its parent IRQ handler while holding the IRQs disabled. The GPIO irqchip will then end up calling something like this sequence in its interrupt handler:

```

static void gpio_irq_handler()
{
    chained_irq_enter(...);
    generic_handle_irq(...);
    chained_irq_exit(...);
}

```

In the code snippet below, extracted from `drivers/pinctrl/bcm/pinctrl-bcm2835.c`, you can see how the interrupt chip functionality is integrated in the GPIO chip, then the GPIO chip is registered with the kernel using the `gpiochip_add_data()` function.

```

struct gpio_irq_chip *girq;

[...]

girq = &pc->gpio_chip.irq;
girq->chip = &bcm2835_gpio_irq_chip;
girq->parent_handler = bcm2835_gpio_irq_handler;
girq->num_parents = BCM2835_NUM_IRQS;
girq->parents = devm_kcalloc(dev, BCM2835_NUM_IRQS,
                             sizeof(*girq->parents),
                             GFP_KERNEL);

if (!girq->parents)
return -ENOMEM;
/*
 * Use the same handler for all groups: this is necessary
 * since we use one gpiochip to cover all lines - the
 * irq handler then needs to figure out which group and
 * bank that was firing the IRQ and look up the per-group
 * and bank data.
 */
for (i = 0; i < BCM2835_NUM_IRQS; i++)
    girq->parents[i] = irq_of_parse_and_map(np, i);
girq->default_type = IRQ_TYPE_NONE;
girq->handler = handle_level_irq;

err = gpiochip_add_data(&pc->gpio_chip, pc);
if (err) {
    dev_err(dev, "could not add GPIO chip\n");
    return err;
}

```

The `gpiochip_add_data()` function will call `gpiochip_add_irqchip()`, which in turn calls `irq_set_chained_handler_and_data()`, which sets a highlevel chained flow handler and its data for a given IRQ. See the code snippet below, extracted from the `gpiochip_add_irqchip()` function:

```

if (gpiochip->irq.parent_handler) {
    void *data = gpiochip->irq.parent_handler_data ?: gpiochip;

    for (i = 0; i < gpiochip->irq.num_parents; i++) {
        /*
         * The parent IRQ chip is already using the chip_data
         * for this IRQ chip, so our callbacks simply use the
         * handler_data.
         */
        irq_set_chained_handler_and_data(gpiochip->irq.parents[i],
                                         gpiochip->irq.parent_handler,
                                         data);
    }
}

```

The `gpiochip_add_irqchip()` function will also call `irq_domain_add_simple()`, which registers an `irq_domain` structure and maps a range of IRQs.

There is a parent handler per GPIO bank interrupt. In the `bcm2835_gpio_irq_handler()` function, there are calls to `bcm2835_gpio_irq_handle_bank()`, which in turn calls `generic_handle_irq()`. The `generic_handle_irq()` wrapper function will call the interrupt handler of each of the GPIO device drivers that is requesting a GPIO interrupt by using the `request_irq()` function. The parameter of the `generic_handle_irq()` function is the Linux IRQ number, which is obtained from a hwirq number by using the `irq_linear_revmap()` function.

```
static void bcm2835_gpio_irq_handler(struct irq_desc *desc)
{
    struct gpio_chip *chip = irq_desc_get_handler_data(desc);
    struct bcm2835_pinctrl *pc = gpiochip_get_data(chip);
    struct irq_chip *host_chip = irq_desc_get_chip(desc);
    int irq = irq_desc_get_irq(desc);
    int group;
    int i;

    for (i = 0; i < BCM2835_NUM_IRQS; i++) {
        if (chip->irq.parents[i] == irq) {
            group = i;
            break;
        }
    }
    /* This should not happen, every IRQ has a bank */
    if (i == BCM2835_NUM_IRQS)
        BUG();

    chained_irq_enter(host_chip, desc);

    switch (group) {
    case 0: /* IRQ0 covers GPIOs 0-27 */
        bcm2835_gpio_irq_handle_bank(pc, 0, 0xffffffff);
        break;
    case 1: /* IRQ1 covers GPIOs 28-45 */
        bcm2835_gpio_irq_handle_bank(pc, 0, 0xf0000000);
        bcm2835_gpio_irq_handle_bank(pc, 1, 0x00003fff);
        break;
    case 2: /* IRQ2 covers GPIOs 46-53 */
        bcm2835_gpio_irq_handle_bank(pc, 1, 0x003fc000);
        break;
    }

    chained_irq_exit(host_chip, desc);
}

static void bcm2835_gpio_irq_handle_bank(struct bcm2835_pinctrl *pc,
                                         unsigned int bank, u32 mask)
{
    unsigned long events;
    unsigned offset;
    unsigned gpio;

    events = bcm2835_gpio_rd(pc, GPEDS0 + bank * 4);
```

```

    events &= mask;
    events &= pc->enabled_irq_map[bank];
    for_each_set_bit(offset, &events, 32) {
        gpio = (32 * bank) + offset;
        generic_handle_irq(irq_linear_revmap(pc->gpio_chip.irq.domain,
                                              gpio));
    }
}

```

2. **NESTED THREADED GPIO irqchips:** These are off-chip GPIO expanders and any other GPIO irqchip residing on the other side of a sleeping bus. Of course, such drivers that need slow bus traffic to read out the IRQ status and similar, traffic which may in turn incur other IRQs to happen, cannot be handled in a quick IRQ handler with IRQs disabled. Instead, they need to spawn a thread and mask the parent IRQ line until the interrupt is handled by the driver.

To help out in handling the setup and management of GPIO irqchips and the associated irqdomain and resource allocation callbacks, the gpiolib has some helpers that can be enabled by selecting the GPIOLIB_IRQCHIP kconfig symbol. These are `gpiochip_irqchip_add()` and `gpiochip_set_chained_irqchip()`.

The `gpiochip_irqchip_add_nested()` function adds a nested irqchip to a gpiochip. This function takes as a parameter the `handle_simple_irq` flow handler, which handles simple interrupts sent from a demultiplexing interrupt handler or coming from hardware where no interrupt hardware control is necessary. The interrupt handler for the GPIO child driver (which requests a GPIO interrupt by using the `devm_request_threaded_irq()` function) will be called inside of a new thread created by the `handle_nested_irq()` function, which is called inside the interrupt handler of the GPIO irqchip driver.

In the LAB 7.4, you will develop a driver for an off-chip GPIO expander with interrupt capabilities that uses the `gpiochip_irqchip_add_nested()` function. This way of adding a nested cascaded irqchip to a gpiochip is still available in kernel 5.4, but the preferred way to set up the helpers today is to fill in the `gpio_irq_chip` structure that is inside `struct gpio_chip` before adding `gpio_chip` to the kernel. In the LAB 7.5, you will develop a driver for the same GPIO expander of the LAB 7.4, but you will use this new method.

The following is a typical example of a nested threaded GPIO irqchip that uses the `gpio_irq_chip` method:

```

/* Typical state container with dynamic irqchip */
struct my_gpio {
    struct gpio_chip gc;
    struct irq_chip irq;
};

int irq; /* from platform etc */

```

```

struct my_gpio *g;
struct gpio_irq_chip *girq;

/* Set up the irqchip dynamically */
g->irq.name = "my_gpio_irq";
g->irq.irq_ack = my_gpio_ack_irq;
g->irq.irq_mask = my_gpio_mask_irq;
g->irq.irq_unmask = my_gpio_unmask_irq;
g->irq.irq_set_type = my_gpio_set_irq_type;

ret = devm_request_threaded_irq(dev, irq, NULL, irq_thread_fn,
                               IRQF_ONESHOT, "my-chip", g);
if (ret < 0)
    return ret;

/* Get a pointer to the gpio_irq_chip */
girq = &g->gc.irq;
girq->chip = &g->irq;
/* This will let us handle the parent IRQ in the driver */
girq->parent_handler = NULL;
girq->num_parents = 0;
girq->parents = NULL;
girq->default_type = IRQ_TYPE_NONE;
girq->handler = handle_bad_irq;

return devm_gpiochip_add_data(dev, &g->gc, g);

```

Device Tree interrupt handling

Unlike address range translation, which follows the natural structure of the tree, interrupt signals can originate from and terminate on any device in a machine. Unlike device addressing, which is naturally expressed in the Device Tree, interrupt signals are expressed as links between nodes independent of the tree. Four properties are used to describe interrupt connections:

1. The **interrupt-controller** property is an empty property, declaring a node as a device that receives interrupt signals.
2. The **interrupt-cells** property is a property of the interrupt controller node that indicates the number of cells in the interrupts property for the child device nodes, for example, the GPIO controller (declared as `gpio` in the Device Tree) of the BCM2837 SoC is also an interrupt parent controller for GPIOx signals and indicates two cells in its `#interrupts-cells` property for the interrupts property of its child device nodes. It is similar to the `#address-cells` and `#size-cells` properties.
3. The **interrupt-parent** property is a property of a device node containing a phandle to the interrupt controller that it is attached to. Nodes that do not have an `interrupt-parent` property can also inherit the property of their parent node.

4. The **interrupts** property is a property of a device node containing a list of interrupt specifiers, one for each interrupt output signal on the device.

See below a code snippet of the BCM2837 gpio controller node (declared in arch/arm/boot/dts/bcm283x.dtsi in the kernel source tree):

```
gpio: gpio@7e200000 {
    compatible = "brcm,bcm2835-gpio";
    reg = <0x7e200000 0xb4>;

    interrupts = <2 17>, <2 18>, <2 19>, <2 20>;

    gpio-controller;
    #gpio-cells = <2>;

    interrupt-controller;
    #interrupt-cells = <2>;

    [...]
};
```

See below the ADXL345 node for the accelerometer driver that will be developed in the LAB 10.3. The interrupt-parent property contains a phandle to the gpio controller node above. The interrupts property contains two interrupt specifiers, as it was indicated with the interrupt-cells property of its parent gpio node.

```
Accel: ADXL345@0 {
    compatible = "arrow,adxl345";
    spi-max-frequency = <5000000>;
    spi-cpol;
    spi-cpha;
    reg = <0>;
    pinctrl-0 = <&accel_int_pin>;
    int-gpios = <&gpio 23 0>;
    interrupts = <23 1>;
    interrupt-parent = <&gpio>;

};
```

Requesting interrupts in Linux device drivers

Interrupts are scheduled by the CPU and run asynchronously. The kernel could be in any state when an interrupt occurs, so the interrupt context cannot access the user buffers and cannot sleep. Handlers can't run actions that may sleep because there is nothing to resume their execution.

As with other resources, a driver must gain access to an interrupt line before it can use it and release it at the end of the execution. In Linux, the request to obtain and release an interrupt is done by using the `request_irq()` and `free_irq()` functions. To perform this task is recommended to use the `devm_request_irq()` managed API for automatic freeing at device or module release time. The `devm_request_irq()` function is defined as follows:

```
devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
                unsigned long irqflags, const char *devname, void *dev_id)
{
    return devm_request_threaded_irq(dev, irq, handler, NULL, irqflags,
                                    devname, dev_id);
}
```

To allocate an interrupt line, you will call `devm_request_irq()`. This function allocates interrupt resources and enables the interrupt line and IRQ handling. When calling this function, you must specify as parameters: a pointer (`dev`) to the device structure, the Linux IRQ number (`irq`), a handler that will be called when the interrupt is generated, flags that will instruct the kernel about the desired behaviour (`irqflags`), the name of the device using this interrupt (`devname`), and a pointer (`dev_id`) that can be configured at any value. Usually, `dev_id` will be a pointer to the device driver's private data. The value that `devm_request_irq()` returns is 0 if the entry was successful or a negative error code indicating the reason for the failure. A typical value is `-EBUSY`, which means that the interrupt was already requested by another device driver.

The main task of an **interrupt handler** is to provide feedback to the processor about the interrupt reception and to read or write data according to the meaning of the interrupt being serviced. The hardware will replay the interrupt (interrupt flood) or won't generate other interrupt until you acknowledge it. The method of acknowledging an interrupt can vary from reading an interrupt controller register, reading the content of a register, or clearing an "interrupt-pending" bit. Some processors have an interrupt acknowledge signal that takes care of this automatically in hardware. The handler function is executed in interrupt context, which means that you can't call blocking APIs such as `mutex_lock()` or `msleep()`. You must also avoid doing a lot of work in the interrupt handler and instead use deferred work if needed. The function prototype is shown below:

```
irqreturn_t (*handler)(int irq_no, void *dev_id);
```

The interrupt handler function receives as parameters the Linux IRQ number of the interrupt (`irq_no`) and the pointer to the private data (`dev_id`) which was sent to `request_irq()` when the interrupt was requested. The interrupt handling routine must return a value with a type of

typedef irqreturn_t. There are three valid values: IRQ_NONE, IRQ_HANDLED and IRQ_WAKE_THREAD. The device driver must return IRQ_NONE if it notices that the interrupt has not been generated by the device it is in charge. Otherwise, the device driver must return IRQ_HANDLED if the interrupt can be handled directly from the interrupt context or IRQ_WAKE_THREAD to schedule the running of the process context processing function.

```
/*
 * enum irqreturn
 * @IRQ_NONE      interrupt was not from this device or was not handled
 * @IRQ_HANDLED   interrupt was handled by this device
 * @IRQ_WAKE_THREAD handler requests to wake the handler thread
 */
enum irqreturn {
    IRQ_NONE      = (0 << 0),
    IRQ_HANDLED   = (1 << 0),
    IRQ_WAKE_THREAD = (1 << 1),
};

typedef enum irqreturn irqreturn_t;
#define IRQ_RETVAL(x)((x) ? IRQ_HANDLED : IRQ_NONE)
```

Your driver should support interrupt sharing whenever this is possible. It is possible if and only if your driver can detect whether your hardware has triggered the interrupt or not. The argument `dev_id` is a sort of client data; this argument is passed to the `devm_request_irq()` function, and the same pointer is passed back as an argument to the handler when the interrupt happens. You usually pass a pointer to your private device data structure in `dev_id`, so you don't need any extra code in the interrupt handler to find out which device is in charge of the current interrupt event. If the handler found that its device did, it should return `IRQ_HANDLED`. If the driver detects that it was not your hardware that caused the interrupt, it will do nothing and returns `IRQ_NONE`, allowing the kernel to call the next interrupt handler.

LAB 7.1: "button interrupt device" module

Throughout the upcoming lab, you will implement your first driver that manages an interrupt. You will use a pushbutton as an interrupt key. The driver will handle button presses. Each time you press the button, an interrupt will be generated and handled by a platform driver.

LAB 7.1 hardware description

For this lab, you will use the button of the MikroElektronika Button R click board. You can see the board at <https://www.mikroe.com/button-r-click>. You can download the schematics from that link or from the GitHub repository of this book. Connect the GPIO23 pin of the Raspberry Pi to the INT pin of the Button R click board.

LAB 7.1 Device Tree description

The GPIO23 pin will be multiplexed in the DT as a GPIO input with internal pull-down enabled. When the button is pressed, the GPIO input value is set to Vcc, then when it is released, the input value is set to GND, generating an interrupt if the `IRQF_TRIGGER_FALLING` flag was passed to the `request_irq()` function. Open and modify the `bcm2710-rpi-3-b.dts` Device Tree file by adding the following code in bold:

```
&gpio {
    spi0_pins: spi0_pins {
        brcm,pins = <9 10 11>;
        brcm,function = <4>; /* alt0 */
    };

    [...]

    key_pin: key_pin {
        brcm,pins = <23>;
        brcm,function = <0>; /* Input */
        brcm,pull = <1>; /* Pull down */
    };
};

&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };

    [...]

    int_key {
        compatible = "arrow,intkey";
        pinctrl-names = "default";
        pinctrl-0 = <&key_pin>;
        gpios = <&gpio 23 0>;
        interrupts = <23 1>;
        interrupt-parent = <&gpio>;
    };
};
```

LAB 7.1 code description of the "button interrupt device" module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/interrupt.h>
#include <linux/gpio/consumer.h>
#include <linux/miscdevice.h>
```

2. For teaching purposes, in the probe() function, you are going to obtain the Linux IRQ number in two different ways. The first method obtains the GPIO descriptor from the gpios property of the int_key DT node by using the devm_gpiod_get() function, then the Linux IRQ number corresponding to the given GPIO is returned by using the function gpiod_to_irq(), which takes the GPIO descriptor as a parameter. The second method uses the platform_get_irq() function, which gets the hwirq number from the interrupts property of the int_key DT node, then returns the Linux IRQ number.

In the probe() function, you will call devm_request_irq() to allocate the interrupt line. When calling this function, you must specify as parameters: a pointer to the device structure, the Linux IRQ number (irq), a handler (hello_keys_isr) that will be called when the interrupt is generated, a flag (IRQF_TRIGGER_FALLING) that will instruct the kernel about the desired interrupt behaviour, the name (HELLO_KEYS_NAME) of the device using this interrupt, and a pointer that can be configured at any value. In this driver, dev_id will point to your device structure.

```
static int __init my_probe(struct platform_device *pdev)
{
    int ret_val, irq;
    struct gpio_desc *gpio;
    struct device *dev = &pdev->dev;

    /* First method to get the virtual linux IRQ number */
    gpio = devm_gpiod_get(dev, NULL, GPIOD_IN);
    irq = gpiod_to_irq(gpio);

    /* Second method to get the virtual Linux IRQ number */
    irq = platform_get_irq(pdev, 0);

    devm_request_irq(dev, irq, hello_keys_isr,
                     IRQF_TRIGGER_FALLING,
                     HELLO_KEYS_NAME, dev);

    misc_register(&helloworld_miscdevice);

    return 0;
}
```

3. Write the interrupt handler. In this driver, an interrupt will be generated and handled (a message will be printed out to the console) each time you press a button. In the handler, you will recover the device structure, which is used as a parameter in the `dev_info()` function.

```
static irqreturn_t hello_keys_isr(int irq, void *data)
{
    struct device *dev = data;
    dev_info(dev, "interrupt received. key: %s\n", HELLO_KEYS_NAME);
    return IRQ_HANDLED;
}
```

4. Declare a list of devices supported by the driver:

```
static const struct of_device_id my_of_ids[] = {
    { .compatible = " arrow,intkey" },
    {}
};
MODULE_DEVICE_TABLE(of, my_of_ids);
```

5. Add a `platform_driver` structure that will be registered to the platform bus:

```
static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "intkey",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};
```

6. Register your driver with the platform bus:

```
module_platform_driver(my_platform_driver);
```

7. Create a new `int_rpi3_key.c` file in the `linux_5.4_rpi3_drivers` folder, and add `int_rpi3_key.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

8. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

9. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 7-1: int_rpi3_key.c

```

#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/interrupt.h>
#include <linux/gpio/consumer.h>
#include <linux/miscdevice.h>

static char *HELLO_KEYS_NAME = "PB_KEY";

/* Interrupt handler */
static irqreturn_t hello_keys_isr(int irq, void *data)
{
    struct device *dev = data;
    dev_info(dev, "interrupt received. key: %s\n", HELLO_KEYS_NAME);
    return IRQ_HANDLED;
}

static struct miscdevice helloworld_miscdevice = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "mydev",
};

static int __init my_probe(struct platform_device *pdev)
{
    int ret_val, irq;
    struct gpio_desc *gpio;
    struct device *dev = &pdev->dev;

    dev_info(dev, "my_probe() function is called.\n");

    /* First method to get the virtual linux IRQ number */
    gpio = devm_gpiod_get(dev, NULL, GPIOD_IN);
    if (IS_ERR(gpio)) {
        dev_err(dev, "gpio get failed\n");
        return PTR_ERR(gpio);
    }
    irq = gpiod_to_irq(gpio);
    if (irq < 0)
        return irq;
    dev_info(dev, "The IRQ number is: %d\n", irq);

    /* Second method to get the virtual Linux IRQ number */
    irq = platform_get_irq(pdev, 0);
    if (irq < 0){
        dev_err(dev, "irq is not available\n");
        return -EINVAL;
    }
    dev_info(dev, "IRQ_using_platform_get_irq: %d\n", irq);

    /* Allocate the interrupt line */
    ret_val = devm_request_irq(dev, irq, hello_keys_isr,
                              IRQF_TRIGGER_FALLING, HELLO_KEYS_NAME, dev);

```

```

    if (ret_val) {
        dev_err(dev, "Failed to request interrupt %d, error %d\n", irq, ret_val);
        return ret_val;
    }

    ret_val = misc_register(&helloworld_miscdevice);
    if (ret_val != 0)
    {
        dev_err(dev, "could not register the misc device mydev\n");
        return ret_val;
    }

    dev_info(dev, "mydev: got minor %i\n", helloworld_miscdevice.minor);
    dev_info(dev, "my_probe() function is exited.\n");

    return 0;
}

static int __exit my_remove(struct platform_device *pdev)
{
    dev_info(&pdev->dev, "my_remove() function is called.\n");
    misc_deregister(&helloworld_miscdevice);
    dev_info(&pdev->dev, "my_remove() function is exited.\n");

    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,intkey"},
    {}},
};

MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "intkey",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};

module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a button INT platform driver");

```

int_rpi3_key.ko demonstration

Load the module:

```
root@raspberrypi:/home/pi# insmod int_rpi3_key.ko
int_rpi3_key: loading out-of-tree module taints kernel.
intkey soc:int_key: my_probe() function is called.
intkey soc:int_key: The IRQ number is: 166
intkey soc:int_key: IRQ_using_platform_get_irq: 166
intkey soc:int_key: mydev: got minor 60
intkey soc:int_key: my_probe() function is exited.
```

Check Raspberry Pi interrupts. See the linux IRQ number (166) with hwirq number (23) for the **pinctrl-bcm2835** controller:

```
root@raspberrypi:/home/pi# cat /proc/interrupts
```

	CPU0	CPU1	CPU2	CPU3			
17:	201	0	0	0	ARMCTRL-level	1 Edge	3f
00b880.mailbox							
18:	718	0	0	0	ARMCTRL-level	2 Edge	VC
HIQ doorbell							
40:	0	0	0	0	ARMCTRL-level	48 Edge	bc
[...]							
162:	1749	2514	1884	1764	bcm2836-timer	1 Edge	ar
ch_timer							
165:	0	0	0	0	bcm2836-pmu	9 Edge	arm-
pmu							
166:	0	0	0	0	pinctrl-bcm2835	23 Edge	
PB_KEY							
FIQ: usb_fiq							
IPI0:	0	0	0	0	CPU wakeup interrupts		
IPI1:	0	0	0	0	Timer broadcast interrupts		
IPI2:	2987	6766	4092	3772	Rescheduling interrupts		
IPI3:	448	662	1605	968	Function call interrupts		
IPI4:	0	0	0	0	CPU stop interrupts		
IPI5:	93	110	22	42	IRQ work interrupts		
IPI6:	0	0	0	0	completion interrupts		
Err: 0							

Press the button of the Button R click board to generate interrupts:

```
root@raspberrypi:/home/pi#
intkey soc:int_key: interrupt received. key: PB_KEY
intkey soc:int_key: interrupt received. key: PB_KEY
intkey soc:int_key: interrupt received. key: PB_KEY
```

Remove the module:

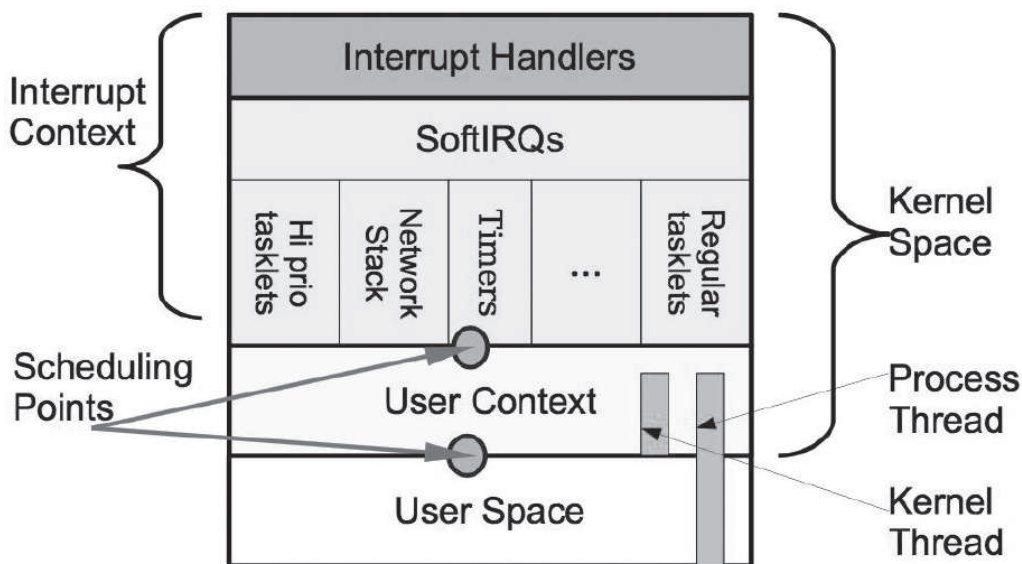
```
root@raspberrypi:/home/pi# rmmod int_rpi3_key.ko
intkey int_key: my_remove() function is called.
intkey int_key: my_remove() function is exited
```

Deferred work

The Linux kernel performs operations in two contexts:

1. **Process context:** Process context is the mode of operation the kernel is in while it is executing on behalf of a user process, for example, executing a system call kernel service routine. Also, the deferred work scheduled by workqueues and threaded interrupts is said to be executed in process context; these kernel threads run in kernel space process context but do not represent any user process. The code executing in process context is able to block.
2. **Interrupt context:** On request from a hardware interrupt controller (asynchronously). This special context is also called "atomic context" because code executing in this context is unable to block. On the other hand, interrupts are not schedulable. They occur and execute the interrupt handler spawning its own context. Softirqs, tasklets and timers are running in interrupt context, which means that they cannot call blocking functions.

Deferred work allows one to schedule code to be executed to a later point. This scheduled code can run either in process context using **workqueues** or **threaded interrupts**, both methods using kernel threads, or in interrupt context using **softirqs**, **tasklets** and **timers**. Work queues and bottom-half of threaded irqs are implemented on top of kernel threads that are able to block, and tasklets and timers are implemented on top of softirqs that cannot call block functions.



Deferred work is used to complement the interrupt handler functionality, since interrupts have important requirements and limitations:

- The execution time of the interrupt handler must be as small as possible.
- In interrupt context you can not use blocking calls.

When using the deferred work you can perform the minimum timing-sensitive required work in the interrupt handler and schedule an asynchronous action from the interrupt handler to run at a later time when the interrupts are enabled. This deferred work used in interrupts is also known as bottom-half, since its purpose is to execute the rest of the action from an interrupt handler (top-half). The **top-half** does what needs to be done immediately, the time critical stuff. Basically, the top-half itself is the interrupt handler. The top-half should complete as quickly as possible since all interrupts are disabled and schedules a bottom half to handle the hard processing. The **bottom-half** does the rest of the processing that has been deferred - the time-dependent, less critical actions. It is signaled by the ISR. A bottom-half is used to process data, letting the top-half to deal with new incoming interrupts. Interrupts are enabled when a bottom-half runs. Interrupts can be disabled if necessary, but generally this should be avoided as this goes against the basic purpose of having a bottom-half - processing data while listening for new interrupts. Interrupt bottom halves are implemented in Linux as softirqs and tasklets in interrupt context, or via threaded irq's in process context.

Softirqs

Softirqs run in interrupt context and are indicated to execute the most timing-critical and important bottom-half processing work that does not need to sleep. They are executed once all interrupt handlers have completed and can be preempted by any top half interrupt. Softirqs can not be used by device drivers, as they are reserved for various kernel subsystems; there is a fixed number of softirqs defined at compile time. For the current kernel version, the following types are defined:

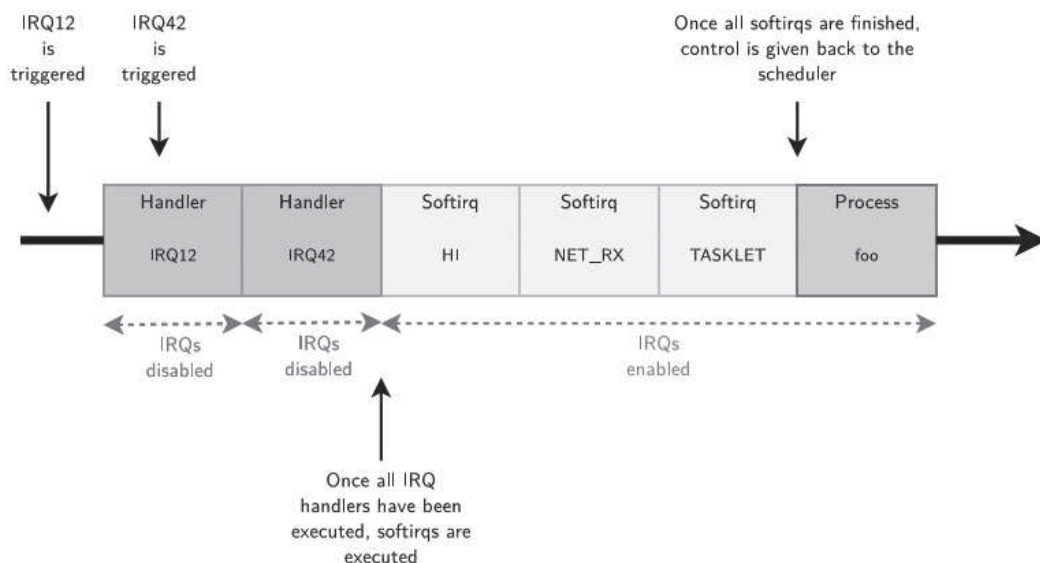
```
enum {
    HI_SOFTIRQ = 0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ,
    NR_SOFTIRQS
};
```

Each type has a specific purpose:

- `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` -- Running tasklets.
- `TIMER_SOFTIRQ` -- Running timers.
- `NET_TX_SOFTIRQ` and `NET_RX_SOFTIRQ` -- Used by the networking subsystem.
- `BLOCK_SOFTIRQ` -- Used by the IO subsystem.
- `BLOCK_IOPOLL_SOFTIRQ` -- Used by the IO subsystem to increase performance when the iopoll handler is invoked.
- `SCHED_SOFTIRQ` -- Load balancing.
- `HRTIMER_SOFTIRQ` -- Implementation of high precision timers.
- `RCU_SOFTIRQ` -- Implementation of RCU type mechanisms.

The highest priority is the `HI_SOFTIRQ` type softirqs, followed in order by the other softirqs defined. `RCU_SOFTIRQ` has the lowest priority.

Softirqs are running in interrupt context, which means that they cannot call blocking functions. If the softirq handler requires calls to such functions, work queues can be scheduled to execute these blocking calls.



Tasklets

A tasklet is a bottom-half mechanism running in interrupt context built on top of softirqs. The main difference between softirqs and tasklets are that tasklets can be allocated dynamically and thus they can be used by device drivers. A tasklet is represented by a tasklet structure and as many other kernel structures it needs to be initialized before being used.

Tasklets are executed within the HI and TASKLET softirqs. They are executed with all interrupts enabled, but a given tasklet is guaranteed to execute on a single CPU at a time. A pre-initialized tasklet can be defined as follows:

```
void handler(unsigned long data);
DECLARE_TASKLET(tasklet, handler, data);
DECLARE_TASKLET_DISABLED(tasklet, handler, data);
```

If you want to initialize the tasklet manually use the `tasklet_init()` function. A tasklet is simply implemented as a function. Tasklets can easily be used by individual device drivers, as opposed to softirqs.

```
void handler(unsigned long data);
struct tasklet_struct tasklet;
tasklet_init(&tasklet, handler, data);
```

The interrupt handler can schedule the tasklet execution using the following functions:

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

When using `tasklet_schedule()`, a `TASKLET_SOFTIRQ` softirq is scheduled, and all tasklets scheduled are run. For `tasklet_hi_schedule()`, a `HI_SOFTIRQ` softirq is scheduled.

Timers

Timers are a type of deferred work running in interrupt context and built on top of softirqs, very often used in Linux drivers. They are defined by the `timer_list` structure. To be used, a timer must first be initialized by calling `setup_timer()`:

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

The previous function initializes the internal fields of the `timer_list` structure and associates a function as the timer handler.

Scheduling a timer is done with `mod_timer()`:

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

Where `expires` parameter is the time (in the future) to run the handler function. The function can be used to schedule or reschedule a timer inside the handler function.

The time unit for the timers is **jiffie**. The absolute value of a jiffie is dependent on the platform, and it can be found by using the `HZ` macro, which defines the number of jiffies for 1 second. To convert between jiffies (`jiffies_value`) and seconds (`seconds_value`), the following formulas are used:

```
jiffies_value = seconds_value * HZ;
seconds_value = jiffies_value / HZ;
```

The kernel maintains a counter that contains the number of jiffies since the last boot, which can be accessed via the `jiffies` global variable or macro. You can use it to calculate a time in the future for timers.

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

unsigned long current_jiffies, next_jiffies;
unsigned long seconds = 1;

current_jiffies = jiffies;
next_jiffies = jiffies + seconds * HZ;
```

To stop a timer, use `del_timer()` and `del_timer_sync()`. A frequent mistake in using timers is that you forget to turn off timers. For example, before removing a module, you must stop the timers because if a timer expires after the module is removed, the handler function will no longer be loaded into the kernel, and a kernel oops will be generated.

You can see below the code of a driver that blinks an LED every second by using a timer deferred work. You can change the blinking period from user space by using the `period sysfs` entry. You can test the driver using a Raspberry Pi 3 board and the Color click™ accessory board with the HW configuration of the LAB 5.2.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/io.h>
#include <linux/timer.h>
#include <linux/device.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>

#define BCM2710_PERI_BASE    0x3F000000
#define GPIO_BASE            BCM2710_PERI_BASE + 0x2000000

struct GpioRegisters
{
    uint32_t GPFSEL[6];
    uint32_t Reserved1;
    uint32_t GPSET[2];
    uint32_t Reserved2;
    uint32_t GPCLR[2];
```

```

};

static struct GpioRegisters *s_pGpioRegisters;

static void SetGPIOFunction(int GPIO, int functionCode)
{
    int registerIndex = GPIO / 10;
    int bit = (GPIO % 10) * 3;

    unsigned oldValue = s_pGpioRegisters->GPFSEL[registerIndex];
    unsigned mask = 0b111 << bit;
    pr_info("Changing function of GPIO%d from %x to %x\n", GPIO,
            (oldValue >> bit) & 0b111, functionCode);
    s_pGpioRegisters->GPFSEL[registerIndex] =
        (oldValue & ~mask) | ((functionCode << bit) & mask);
}

static void SetGPIOOutputValue(int GPIO, bool outputValue)
{
    if (outputValue)
        s_pGpioRegisters->GPSET[GPIO / 32] = (1 << (GPIO % 32));
    else
        s_pGpioRegisters->GPCLR[GPIO / 32] = (1 << (GPIO % 32));
}

static struct timer_list s_BlinkTimer;
static int s_BlinkPeriod = 1000;
static const int LedGpioPin = 27;

static void BlinkTimerHandler(unsigned long unused)
{
    static bool on = false;
    on = !on;
    SetGPIOOutputValue(LedGpioPin, on);
    mod_timer(&s_BlinkTimer, jiffies + msecs_to_jiffies(s_BlinkPeriod));
}

static ssize_t set_period(struct device* dev,
                          struct device_attribute* attr,
                          const char* buf,
                          size_t count)
{
    long period_value = 0;
    if (kstrtol(buf, 10, &period_value) < 0)
        return -EINVAL;
    if (period_value < 10)
        return -EINVAL;

    s_BlinkPeriod = period_value;
    return count;
}

static DEVICE_ATTR(period, S_IWUSR, NULL, set_period);

```

```

static struct miscdevice led_miscdevice = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "ledred",
};

static int __init my_probe(struct platform_device *pdev) {
    int result, ret_val;
    struct device *dev = &pdev->dev;
    dev_info(dev, "platform_probe enter\n");
    s_pGpioRegisters = (struct GpioRegisters *)devm_ioremap(dev, GPIO_BASE,
                                                            sizeof(struct GpioRegisters));

    SetGPIOFunction(LedGpioPin, 0b001); /* Configure the pin as output */

    setup_timer(&s_BlinkTimer, BlinkTimerHandler, 0);
    result = mod_timer(&s_BlinkTimer, jiffies + msecs_to_jiffies(s_BlinkPeriod));

    ret_val = device_create_file(&pdev->dev, &dev_attr_period);
    if (ret_val != 0)
    {
        dev_err(dev, "failed to create sysfs entry");
        return ret_val;
    }

    ret_val = misc_register(&led_miscdevice);
    if (ret_val != 0)
    {
        dev_err(dev, "could not register the misc device mydev");
        return ret_val;
    }
    dev_info(dev, "mydev: got minor %i\n", led_miscdevice.minor);

    dev_info(dev, "platform_probe exit\n");
    return 0;
}

static int __exit my_remove(struct platform_device *pdev)
{
    dev_info(&pdev->dev, "platform_remove enter\n");
    misc_deregister(&led_miscdevice);
    device_remove_file(&pdev->dev, &dev_attr_period);
    SetGPIOFunction(LedGpioPin, 0);
    del_timer(&s_BlinkTimer);
    dev_info(&pdev->dev, "platform_remove exit\n");
    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,ledred"},
    {}},
};

static struct platform_driver my_platform_driver = {

```

```

.probe = my_probe,
.remove = my_remove,
.driver = {
    .name = "ledred",
    .of_match_table = my_of_ids,
    .owner = THIS_MODULE,
}
};

module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a blinking led driver");

```

Threaded interrupts

You can find situations where the device driver handling the interrupts can't read the registers of the device in a non-blocking mode (for example, a sensor connected to an I2C or SPI bus whose driver does not guarantee that bus read/write operations are non-blocking). In this situation, you must plan a work-in-process action (work queue, kernel thread) to access the registers of the device. Because such a situation is relatively common, the kernel provides the `request_threaded_irq()` function to write interrupt handling routines running in two phases: a process phase and an interrupt context phase. As with the function `request_irq()`, it is recommended to use the managed API `devm_request_threaded_irq()`. Within the parameters of the function, `handler` is the function running in interrupt context and implements the critical operations, while the `thread_fn` handler runs in process context and implements the rest of the operations. The `devm_request_threaded_irq()` function is defined as follows:

```

/*
 * devm_request_threaded_irq - allocate an interrupt line for a managed device
 * @dev: device to request interrupt for
 * @irq: Interrupt line to allocate
 * @handler: Function to be called when the IRQ occurs
 * @thread_fn: function to be called in a threaded interrupt context. NULL
 * for devices which handle everything in @handler
 * @irqflags: Interrupt type flags
 * @devname: An ascii name for the claiming device
 * @dev_id: A cookie passed back to the handler function
 * Except for the extra @dev argument, this function takes the
 * same arguments and performs the same function as
 * request_threaded_irq(). IRQs requested with this function will be
 * automatically freed on driver detach.
 * If an IRQ allocated with this function needs to be freed
 * separately, devm_free_irq() must be used.
 */
int devm_request_threaded_irq(struct device *dev, unsigned int irq,
                             irq_handler_t handler, irq_handler_t thread_fn,
                             unsigned long irqflags, const char *devname, void *dev_id)

```

```
{
    struct irq_devres *dr;
    int rc;

    dr = devres_alloc(devm_irq_release, sizeof(struct irq_devres), GFP_KERNEL);
    if (!dr)
        return -ENOMEM;

    rc = request_threaded_irq(irq, handler, thread_fn, irqflags, devname, dev_id);
    if (rc) {
        devres_free(dr);
        return rc;
    }
}
```

If the handler parameter is set to NULL, the default primary handler will be executed. This primary handler only returns IRQ_WAKE_THREAD to wake up the associated kernel thread which will execute the thread_fn handler. The thread_fn handler will run with all the interrupts enabled.

The flag parameter can be zero or a bit mask using one or combining some of the flags defined in include/linux/interrupt.h in the kernel source tree. These are some of the most important flags:

- **IRQF_DISABLED:** When this flag is set, all the interrupts are disabled while the interrupt handler is being executed. Another processor core cannot handle a new interrupt that occurs while the current interrupt handler is running. When this flag is disabled, the interrupt handler runs with all the interrupts except their own enabled, so another processor core can handle a new different interrupt while the current interrupt handler is being serviced.
- **IRQF_SHARED:** When this flag is set, the interrupt line can be shared among several interrupt handlers. All of these interrupt handlers will execute until the interrupt responsible is found. When this flag is disabled, only one handler can exist per interrupt line. The request for interrupt will fail if there is already a handler for the requested interrupt.
- **IRQF_ONESHOT:** When this flag is set, the interrupt line will be reactivated after the thread_fn handler (process context) is being executed. When this flag is disabled, the interrupt line will be reactivated after running the handler routine (interrupt context). It is not convenient to use IRQF_ONESHOT with IRQF_SHARED.

Workqueues

Workqueues are another way of deferring work. The base unit with which they work is called **work** and is queued in a **workqueue**. Workqueues defer work into a kernel thread that runs in process context and can sleep unlike softirqs and tasklets. The other alternative to workqueues is kernel threads, although using of workqueues is more convenient and easy to use. A kernel thread called **worker** will be responsible for handling the work queued in the workqueue, dequeuing the work items from the workqueue and executing the functions associated with those items.

A **work item** is a simple data structure that holds a pointer to the function that is to be executed asynchronously. Whenever a user (e.g., a driver or subsystem) wants a function to be executed asynchronously by means of workqueues, it has to set up a work item pointing to that function and queue that work item on a workqueue. Special purpose threads, called worker threads, execute the functions after dequeuing the items, one after the other. If no work is queued, the worker threads become idle.

Worker threads are controlled by worker-pools, which take care of the level of concurrency (the simultaneously running worker threads) and the process management.

Subsystems and drivers can create and queue work items through special workqueue API functions as they see fit. They can influence some aspects of the way the work items are executed by setting flags on the workqueue they are putting the work item on. These flags include things like CPU locality, concurrency limits, priority and more. To get a detailed overview, refer to the API description of the `alloc_workqueue()` function.

The workqueue API offers two types of function interfaces: first, a set of interface routines to instantiate and queue work items onto a global workqueue, which is shared by all kernel subsystems and services, and second, a set of interface routines to set up a new workqueue and queue work items onto it. You will begin to explore workqueue interfaces with macros and functions related to the global shared workqueue.

There are two types of structures associated to the worker thread:

- `work_struct` -- It schedules a task to run at a later time.
- `delayed_work` -- It schedules a task to run after at least a given time interval.

A delayed work uses a timer to run after the specified time interval. The calls with this type of work are similar to those for `work_struct`, but has `_delayed` in the function names. Before using them, a work item must be initialized. There are two types of macros that can be used, one that declares and initializes the work item at the same time, and one that only initializes the work item (and the declaration must be done separately):

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

```
DECLARE_WORK(name, void (*function)(struct work_struct *));
DECLARE_DELAYED_WORK(name, void(*function)(struct work_struct *));

INIT_WORK(struct work_struct *work, void(*function)(struct work_struct *));
INIT_DELAYED_WORK(struct delayed_work *work, void(*function)(struct work_struct *));
```

`DECLARE_WORK()` and `DECLARE_DELAYED_WORK()` declare and initialize a work item, and `INIT_WORK()` and `INIT_DELAYED_WORK()` initialize an already declared work item.

The following code snippet declares and initiates a work item:

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

void my_work_handler(struct work_struct *work);
DECLARE_WORK(my_work, my_work_handler);
```

Or, if you want to initialize the work item separately:

```
void my_work_handler(struct work_struct *work);
struct work_struct my_work;
INIT_WORK(&my_work, my_work_handler);
```

Once declared and initialized, a work instance can be scheduled in the workqueue through `schedule_work()` or `schedule_delayed_work()`. This function enqueues the given work item on the local CPU workqueue, but does not guarantee its execution of it. It returns true if the given work is successfully enqueued, or false if the given work is already found in the workqueue. Once queued, the function associated with the work item is executed on any of the available CPUs by the relevant kworker thread:

```
schedule_work(struct work_struct *work);
schedule_delayed_work(struct delayed_work *work, unsigned long delay);
```

You can wait for a workqueue to complete running all of its work items by calling `flush_scheduled_work()`.

Finally, the following functions can be used to schedule work items on a particular CPU (`schedule_delayed_work_on()`) or on all CPUs (`schedule_on_each_cpu()`):

```
int schedule_delayed_work_on(int cpu, struct delayed_work *work, unsigned long delay);
int schedule_on_each_cpu(void(*function)(struct work_struct *));
```

The following code snippet initializes and schedules a work item:

```
void my_work_handler(struct work_struct *work);
struct work_struct my_work;
INIT_WORK(&my_work, my_work_handler);
schedule_work(&my_work);
```

You can use `flush_scheduled_work()` to wait the termination of a work item.

To be able to access the private data of the driver, you can use `container_of()`, as shown in the following code snippet:

```

struct my_device_data {
    struct work_struct my_work;

    [...]
};

void my_work_handler(struct work_struct *work)
{
    structure my_device_data * my_data;
    my_data = container_of(work, struct my_device_data, my_work);

    [...]
}

```

The handler will run in the context of a kernel thread called `events/x`, where `x` is the processor core number. The kernel will initialize a kernel thread (or a pool of workers) for each processor core present in the system. The above functions use a predefined workqueue (called `events`), and they run in the context of the `events/x` thread, as noted above. Although this is sufficient in most cases, it is a shared resource, and large delays in work item handlers can cause delays for other queue users. For this reason, there are functions for creating additional queues.

A workqueue is represented by the `workqueue_struct` structure. A new workqueue can be created with the following functions:

```

struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);

```

The `create_workqueue()` function uses one thread for each processor in the system, and `create_singlethread_workqueue()` uses a single thread.

To add a task in the new queue, use `queue_work()` or `queue_delayed_work()`:

```

int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue,
                      struct delayed_work *work, unsigned long delay);

```

To wait for all work items to finish, call `flush_workqueue()`:

```

void flush_workqueue(struct workqueue_struct *queue);

```

To destroy the workqueue, call `destroy_workqueue()`:

```

void destroy_workqueue(struct workqueue_struct *queue);

```

The following code snippet declares and initializes an additional workqueue, then declares and initializes a work item and adds it to the queue:

```

void my_work_handler(struct work_struct *work);

struct work_struct my_work;
struct workqueue_struct *my_workqueue;

```

```
my_workqueue = create_singlethread_workqueue("my_workqueue");
INIT_WORK(&my_work, my_work_handler);

queue_work(my_workqueue, &my_work);
```

The following lines of code show how to remove the workqueue:

```
flush_workqueue(my_workqueue);
destroy_workqueue(my_workqueue);
```

Note: Part of the text of the "Deferred Work" section has been extracted from the Linux kernel Documentation. You can find further information in the following links:

https://linux-kernel-labs.github.io/refs/heads/master/labs/deferred_work.html

<https://linux-kernel-labs.github.io/refs/heads/master/lectures/interrupts.html>

<https://www.kernel.org/doc/html/latest/core-api/workqueue.html>

Locking in the kernel

Imagine you have two different kthreads that read the value of a shared counter and have to increment it after reading. It can happen that one of the two kthreads (let's call it kthread1) that is running, reads the value of the shared counter and stores it in a local variable, then before incrementing it, the kthread1 is preempted by the other kthread (let's call it kthread2), which reads the same value of the shared counter and stores it in another local variable. Now, the kthread1 preempts the kthread2 and increments by one the value of its local variable, then updates the shared counter value. Finally, the kthread2 preempts the kthread1, increments the value of its local variable and updates the shared counter. The result is that the value of the counter has only been incremented by one unit after being incremented by the two kthreads. This overlap, where the result depends on the relative timing of multiple tasks, is called a **race condition**. The piece of code containing the concurrency issue is called a critical region. And especially since Linux started running on SMP machines, they became one of the major issues in kernel design and implementation. The solution is to recognize when these simultaneous accesses occur, and use locks to make sure that only one instance can enter the critical region at any time.

There are two main types of kernel locks. The fundamental type is the **spinlock** (declared in `include/asm/spinlock.h` in the kernel source tree), which is a very simple single-holder lock: if you can't get the spinlock, you keep trying (spinning) until you can (disables the preemption in the running core). Spinlocks are very small and fast, and they can be used anywhere.

The second type is a **mutex** (declared in `include/linux/mutex.h` in the kernel source tree); it is like a spinlock, but you may block holding a mutex. If you can't lock a mutex, your task will suspend itself and be woken up when the mutex is released. This means the CPU can do something else while you are waiting.

Locks and uniprocessor kernels

For kernels compiled without `CONFIG_SMP` and without `CONFIG_PREEMPT`, spinlocks do not exist at all. This is an excellent design decision: when no-one else can run at the same time, there is no reason to have a lock.

If the kernel is compiled without `CONFIG_SMP`, but `CONFIG_PREEMPT` is set, then spinlocks simply disable preemption, which is sufficient to prevent any races.

Sharing spinlocks between interrupt and process context

It is possible that a critical section needs to be protected by the same lock in both an interrupt and a non-interrupt (process) execution context in the kernel. In this case, the `spin_lock_irqsave` and `spin_unlock_irqrestore` variants have to be used to protect the critical section. This has the effect of disabling interrupts on the executing processor core. You can see in the following steps what could happen if you just used `spin_lock` in the process context:

1. Process context kernel code acquires the spinlock by using `spin_lock`.
2. While the spinlock is held, an interrupt comes in on the same processor core and executes.
3. Interrupt Service Routing (ISR) tries to acquire the spinlock and spins continuously waiting for it. Process context blocks in the current processor core, and there have been never a chance to run again and free the spinlock.

To avoid this, the process context kernel code will use the `spin_lock_irqsave` function, which has the effect of disabling interrupts on that particular processor core. Both, `spin_lock` and `spin_lock_irqsave` can be called in the ISR, as interrupts are disabled anyway on the executing processor core.

The interrupts are only disabled on the executing processor core while the critical section protected by `spin_lock_irqsave` is being executed. An interrupt can be handled on another processor core, which can access to the shared spinlock inside its ISR; in this situation the process context kernel code is not blocked and can finish executing the locked critical section and release the spinlock while the ISR spins on the spinlock. In the LAB 7.3, you will see the use of shared spinlocks.

Locking in user context

If you have a data structure which is only ever accessed from user context, then you can use a simple mutex to protect it. This is the most trivial case: you initialize the mutex, then you call `mutex_lock_interruptible()` to grab the mutex and `mutex_unlock()` to release it. There is also a `mutex_lock()` function which should be avoided because it will not return if a signal is received.

Sleeping in the kernel

In Linux kernel programming, there are many situations where user processes need to sleep and be woken up when some specific work is being done. The concept of "Sleeping in the kernel" is very well explained in the following excerpt from LDD3 book (<https://lwn.net/Kernel/LDD3/>):

What does it mean for a user process to sleep? When a process is put to sleep, it is marked as being in a special state and removed from the scheduler's run queue. Until something comes along to change that state, the process will not be scheduled on any CPU and, therefore, will not run. A sleeping process has been shunted off to the side of the system, waiting for some future event to happen.

Causing a process to sleep is an easy thing for a Linux device driver to do. There are, however, a couple of rules that you must keep in mind to be able to sleep the process in a safe manner. The first of these rules is: never sleep when you are running in an atomic context. You also cannot sleep if you have disabled interrupts. It is legal to sleep while holding a semaphore, but you should look very carefully at any code that does so. If code sleeps while holding a semaphore, any other thread waiting for that semaphore also sleeps. Another thing to remember with sleeping is that, when you wake up, you never know how long your process may have been out of the CPU or what may have changed in the mean time. You also do not usually know if another process may have been sleeping due to the same event. Other relevant point is that your process cannot sleep unless it is assured that somebody will wake it up. The code doing the waking must also be able to find your process to be able to do its job. Making it possible for your sleeping process to be found is accomplished through a data structure called a wait queue. A wait queue is a list of processes, all waiting for a specific event.

In Linux, a wait queue is used to wait for someone to wake you up when a certain condition is true. You need to declare a structure of type `wait_queue_head_t`. You can define and initialize a wait queue statically:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

or dynamically:

```
wait_queue_head_t my_queue;  
init_waitqueue_head(&my_queue);
```

The `wait_event(wq, condition)` macro (with a few variants) will put the process to sleep (`TASK_UNINTERRUPTIBLE`) until the condition evaluates to true. The condition is checked each time the wq waitqueue is woken up. These are the different variants of the `wait_event` macro:

```
wait_event(queue, condition);  
wait_event_interruptible(queue, condition);  
wait_event_timeout(queue, condition, timeout);  
wait_event_interruptible_timeout(queue, condition, timeout);
```

All the previous functions share the queue parameter. The condition parameter will be evaluated by the macro before and after sleeping; until the condition evaluates to a true value, the process continues to sleep. If you use `wait_event`, your process is put into an uninterruptible sleep. The preferred alternative is `wait_event_interruptible`, which can be interrupted by signals. The timeout versions (`wait_event_timeout` and `wait_event_interruptible_timeout`) wait for a limited time; after that time period (expressed in jiffies) expires, the macros return with a value of zero regardless of how the condition evaluates.

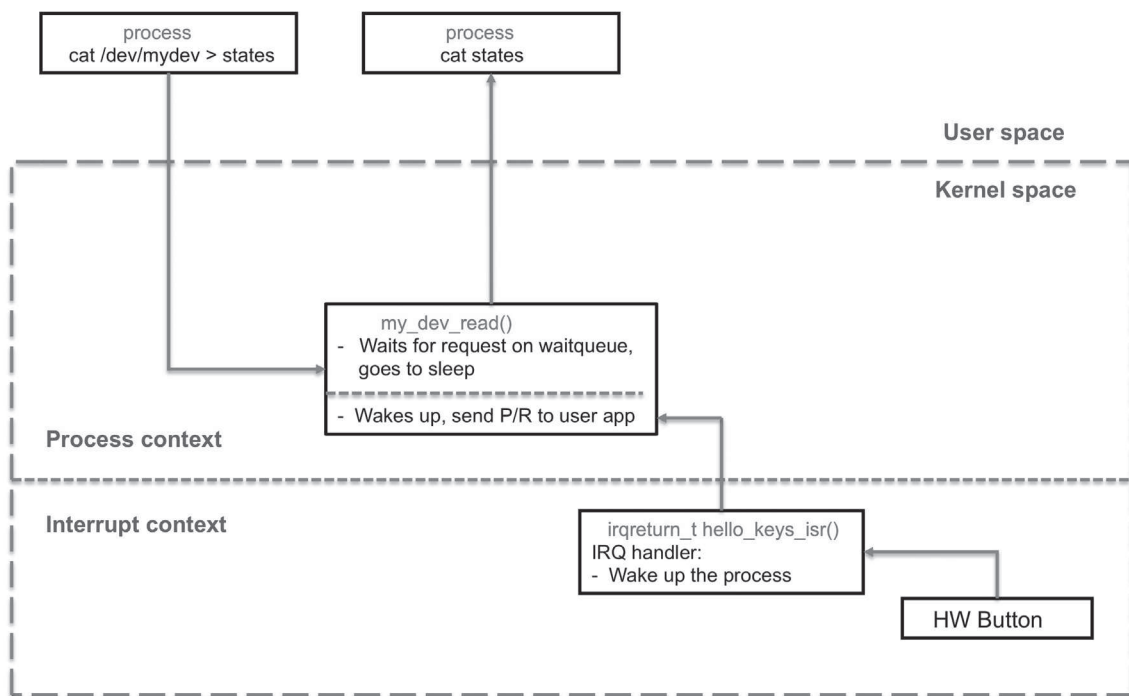
Another different process, or an interrupt handler will wake up the asleep process. The `wake_up()` function awakens processes that have gone to sleep using the same condition variable. This function does not block and may be called from interrupt handlers. You can see below two of its forms:

```
void wake_up(wait_queue_head_t *queue); /* wake_up wakes up all processes waiting on the given queue */
```

```
void wake_up_interruptible(wait_queue_head_t *queue); /* restricts itself to processes performing an interruptible sleep */
```

LAB 7.2: "sleeping device" module

In this LAB 7.2, you are going to develop a kernel module that causes a process to sleep, then wakes it up via an interrupt. You can see the behavior of the driver in the following image. When the user application attempts to read (system call) from the device, the process is put to sleep. Every time you press or release a button, the generated interrupt will wake up the process and the driver's read callback function will send to user space the type of interrupt that was set (Press or Release). Once you have exited the user application, you can read from a file all the interrupts that were generated.



You will keep the same HW configuration used in the previous LAB 7.1.

LAB 7.2 Device Tree description

The GPIO23 pin will be multiplexed in the DT as a GPIO input with internal pull-down enabled. Two interrupts are going to be generated, the first one when the button is pressed and the second one when it is released. You have to set `IRQ_TYPE_EDGE_BOTH` in the interrupts property.

Open and modify the `bcm2710-rpi-3-b.dts` Device Tree file by adding the following code in bold:

```
&gpio {
    spi0_pins: spi0_pins {
        brcm,pins = <9 10 11>;
        brcm,function = <4>; /* alt0 */
    };

    [...]

    key_pin: key_pin {
        brcm,pins = <23>;
        brcm,function = <0>;      /* Input */
        brcm,pull = <1>;         /* Pull down */
    };
};

&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };

    [...]

    int_key_wait {
        compatible = "arrow,intkeywait";
        pinctrl-names = "default";
        pinctrl-0 = <&key_pin>;
        gpios = <&gpio 23 0>;
        interrupts = <23 IRQ_TYPE_EDGE_BOTH>;
        interrupt-parent = <&gpio>;
    };
};
```

LAB 7.2 code description of the "sleeping device" module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/of_gpio.h>
#include <linux/of_irq.h>
#include <linux/uaccess.h>
#include <linux/interrupt.h>
#include <linux/miscdevice.h>
```

2. Create a private structure that will store the button device-specific information. The second field of the private structure is a pointer to the `gpio_desc` structure associated with your button. In this driver, you will handle a char device, so a `miscdevice` structure will be created, initialized and added to your device-specific data structure in the third field. The fourth field of the private structure is a structure of type `wait_queue_head_t` that will be initialized dynamically within the `probe()` function. The last field will store the Linux IRQ number.

```
struct key_priv {
    struct device *dev;
    struct gpio_desc *gpio;
    struct miscdevice int_miscdevice;
    wait_queue_head_t wq_data_available;
    int irq;
};
```

3. In the `probe()` function, a wait queue head is initialized with the line of code `init_waitqueue_head(&priv->wq_data_available)`. You will recover the DT interrupt number by using the same two methods of the LAB 7.1. In the `probe()` function, you will also call `devm_request_irq()` to allocate the interrupt line. When calling this function, you must specify as parameters: a pointer to the device structure, the interrupt number, a handler (`hello_keys_isr`) that will be called when the interrupt is generated, a flag (`IRQF_TRIGGER_RISING` | `IRQF_TRIGGER_FALLING`) that will instruct the kernel about the desired interrupt behaviour, the name (`HELLO_KEYS_NAME`) of the device which uses this interrupt, and a pointer to your private structure.

```
static int __init my_probe(struct platform_device *pdev)
{
    struct key_priv *priv;
    struct device *dev = &pdev->dev;

    /* Allocate new structure representing device */
    priv = devm_kzalloc(dev, sizeof(struct key_priv), GFP_KERNEL);
    priv->dev = dev;
```

```

platform_set_drvdata(pdev, priv);

init_waitqueue_head(&priv->wq_data_available);

/* Get the Linux IRQ number from Device Tree in 2 ways */
priv->gpio = devm_gpiod_get(dev, NULL, GPIOD_IN);
priv->irq = gpiod_to_irq(priv->gpio);

priv->irq = platform_get_irq(pdev, 0);

devm_request_irq(dev, priv->irq, hello_keys_isr,
                 IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                 HELLO_KEYS_NAME, priv);

priv->int_miscdevice.name = "mydev";
priv->int_miscdevice.minor = MISC_DYNAMIC_MINOR;
priv->int_miscdevice.fops = &my_dev_fops;

ret_val = misc_register(&priv->int_miscdevice);
return 0;
}

```

4. Write now the interrupt handler. In this driver, an interrupt will be generated and handled each time you press and release a button. In the handler, you will recover the private structure from the data argument. Once you have retrieved the private structure, you can read the GPIO input value by using the `gpiod_get_value()` function to determine if you have pressed or released the button. After reading the input, you will wake up the process by using the `wake_up_interruptible()` function, which takes as its argument the wait queue head declared in your private structure.

```

static irqreturn_t hello_keys_isr(int irq, void *data)
{
    int val;
    struct key_priv *priv = data;
    dev_info(priv->dev, "interrupt received. key: %s\n", HELLO_KEYS_NAME);

    val = gpiod_get_value(priv->gpio);
    dev_info(priv->dev, "Button state: 0x%08X\n", val);

    if (val == 1)
        hello_keys_buf[buf_wr++] = 'P';
    else
        hello_keys_buf[buf_wr++] = 'R';

    if (buf_wr >= MAX_KEY_STATES)
        buf_wr = 0;

    /* Wake up the process */
    wake_up_interruptible(&priv->wq_data_available);
}

```

```

        return IRQ_HANDLED;
    }

```

5. Create the `my_dev_read()` kernel function, which gets called whenever a user space read operation occurs on the character device file. You will recover the private structure by using the `container_of()` macro. The `wait_event_interruptible()` function puts the user process to sleep into the wait queue, waiting for a specific event. In this function, you will set as a parameter the condition to be evaluated to wake up the process. When the process is woken up, the 'P' or 'R' character (that was stored within the ISR) is sent to user space by using the `copy_to_user()` function.

```

static int my_dev_read(struct file *file, char __user *buff,
                      size_t count, loff_t *off)
{
    int ret_val;
    char ch[2];
    struct key_priv *priv;
    container_of(file->private_data,
                 struct key_priv, int_miscdevice);
    /*
     * Sleep the process.
     * The condition is checked each time the waitqueue is woken up
     */
    wait_event_interruptible(priv->wq_data_available, buf_wr != buf_rd);

    /* Send values to user application */
    ch[0] = hello_keys_buf[buf_rd];
    ch[1] = '\n';
    copy_to_user(buff, &ch, 2);

    buf_rd++;
    if(buf_rd >= MAX_KEY_STATES)
        buf_rd = 0;
    *off+=1;
    return 2;
}

```

6. Declare a list of devices supported by the driver:

```

static const struct of_device_id my_of_ids[] = {
    { .compatible = " arrow,intkeywait"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

```

7. Add a `platform_driver` structure that will be registered to the platform bus:

```

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "intkeywait",
        .of_match_table = my_of_ids,
    }
};

```

```
        .owner = THIS_MODULE,
    }
};
```

8. Register your driver with the platform bus:

```
module_platform_driver(my_platform_driver);
```

9. Create a new `int_rpi3_key_wait.c` file in the `linux_5.4_rpi3_drivers` folder, and add `int_rpi3_key_wait.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

10. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

11. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 7-2: `int_rpi3_key_wait.c`

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/of_gpio.h>
#include <linux/of_irq.h>
#include <linux/uaccess.h>
#include <linux/interrupt.h>
#include <linux/miscdevice.h>
#include <linux/wait.h> /* include wait queue */

#define MAX_KEY_STATES 256

static char *HELLO_KEYS_NAME = "PB_USER";
static char hello_keys_buf[MAX_KEY_STATES];
static int buf_rd, buf_wr;

struct key_priv {
    struct device *dev;
    struct gpio_desc *gpio;
    struct miscdevice int_miscdevice;
    wait_queue_head_t wq_data_available;
    int irq;
};

static irqreturn_t hello_keys_isr(int irq, void *data)
{
    int val;
```

```

struct key_priv *priv = data;
dev_info(priv->dev, "interrupt received. key: %s\n", HELLO_KEYS_NAME);

val = gpiod_get_value(priv->gpio);
dev_info(priv->dev, "Button state: 0x%08X\n", val);

if (val == 1)
    hello_keys_buf[buf_wr++] = 'P';
else
    hello_keys_buf[buf_wr++] = 'R';

if (buf_wr >= MAX_KEY_STATES)
    buf_wr = 0;

/* Wake up the process */
wake_up_interruptible(&priv->wq_data_available);

return IRQ_HANDLED;
}

static int my_dev_read(struct file *file, char __user *buff,
                      size_t count, loff_t *off)
{
    int ret_val;
    char ch[2];
    struct key_priv *priv;

    priv = container_of(file->private_data,
                        struct key_priv, int_miscdevice);

    dev_info(priv->dev, "mydev_read_file entered\n");

    /*
     * Sleep the process.
     * The condition is checked each time the waitqueue is woken up
     */
    ret_val = wait_event_interruptible(priv->wq_data_available, buf_wr != buf_rd);
    if(ret_val)
        return ret_val;

    /* Send values to user application */
    ch[0] = hello_keys_buf[buf_rd];
    ch[1] = '\n';
    if(copy_to_user(buff, &ch, 2)) {
        return -EFAULT;
    }

    buf_rd++;
    if(buf_rd >= MAX_KEY_STATES)
        buf_rd = 0;
    *off+=1;
    return 2;
}

```

```

static const struct file_operations my_dev_fops = {
    .owner = THIS_MODULE,
    .read = my_dev_read,
};

static int __init my_probe(struct platform_device *pdev)
{
    int ret_val;
    struct key_priv *priv;
    struct device *dev = &pdev->dev;

    dev_info(dev, "my_probe() function is called.\n");

    /* Allocate the private structure */
    priv = devm_kzalloc(dev, sizeof(struct key_priv), GFP_KERNEL);
    priv->dev = dev;

    platform_set_drvdata(pdev, priv);

    /* Init the wait queue head */
    init_waitqueue_head(&priv->wq_data_available);

    /* Get the virtual interrupt number from Device Tree using 2 methods */
    priv->gpio = devm_gpiod_get(dev, NULL, GPIOD_IN);
    if (IS_ERR(priv->gpio)) {
        dev_err(dev, "gpio get failed\n");
        return PTR_ERR(priv->gpio);
    }
    priv->irq = gpiod_to_irq(priv->gpio);
    if (priv->irq < 0)
        return priv->irq;
    dev_info(dev, "The IRQ number is: %d\n", priv->irq);

    priv->irq = platform_get_irq(pdev, 0);
    if (priv->irq < 0) {
        dev_err(dev, "irq is not available\n");
        return priv->irq;
    }
    dev_info(dev, "IRQ_using_platform_get_irq: %d\n", priv->irq);

    ret_val = devm_request_irq(dev, priv->irq, hello_keys_isr,
                              IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                              HELLO_KEYS_NAME, priv);
    if (ret_val) {
        dev_err(dev, "Failed to request interrupt %d, error %d\n", priv->irq, ret_val);
        return ret_val;
    }

    priv->int_miscdevice.name = "mydev";
    priv->int_miscdevice.minor = MISC_DYNAMIC_MINOR;
    priv->int_miscdevice.fops = &my_dev_fops;

    ret_val = misc_register(&priv->int_miscdevice);
    if (ret_val != 0)

```

```

{
    dev_err(dev, "could not register the misc device mydev\n");
    return ret_val;
}

dev_info(dev, "my_probe() function is exited.\n");

return 0;
}

static int __exit my_remove(struct platform_device *pdev)
{
    struct key_priv *priv = platform_get_drvdata(pdev);
    dev_info(&pdev->dev, "my_remove() function is called.\n");
    misc_deregister(&priv->int_miscdevice);
    dev_info(&pdev->dev, "my_remove() function is exited.\n");
    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,intkeywait"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "intkeywait",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};
module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a platform driver that sends to user space \
    the number of times you press the switch using INTs");

```

int_rpi3_key_wait.ko demonstration

Load the module:

```

root@raspberrypi:/home/pi# insmod int_rpi3_key_wait.ko
intkeywait soc:int_key_wait: my_probe() function is called.
intkeywait soc:int_key_wait: The IRQ number is: 166
intkeywait soc:int_key_wait: IRQ_using_platform_get_irq: 166
intkeywait soc:int_key_wait: my_probe() function is exited.

```


Sleep the process. Press and release the button of the Button R click board several times:

```
root@raspberrypi:/home/pi# cat /dev/mydev > states
intkeywait soc:int_key_wait: mydev_read_file entered
intkeywait soc:int_key_wait: interrupt received. key: PB_USER
intkeywait soc:int_key_wait: Button state: 0x00000001
intkeywait soc:int_key_wait: mydev_read_file entered
intkeywait soc:int_key_wait: interrupt received. key: PB_USER
intkeywait soc:int_key_wait: Button state: 0x00000000
intkeywait soc:int_key_wait: mydev_read_file entered
intkeywait soc:int_key_wait: interrupt received. key: PB_USER
intkeywait soc:int_key_wait: Button state: 0x00000001
intkeywait soc:int_key_wait: mydev_read_file entered
intkeywait soc:int_key_wait: interrupt received. key: PB_USER
intkeywait soc:int_key_wait: Button state: 0x00000000
intkeywait soc:int_key_wait: mydev_read_file entered
intkeywait soc:int_key_wait: interrupt received. key: PB_USER
intkeywait soc:int_key_wait: Button state: 0x00000001
intkeywait soc:int_key_wait: mydev_read_file entered
intkeywait soc:int_key_wait: interrupt received. key: PB_USER
intkeywait soc:int_key_wait: Button state: 0x00000000
```

Check all the times you pressed and released the button:

```
root@raspberrypi:/home/pi# cat states
P
R
P
R
P
R
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod int_rpi3_key_wait.ko
```

Kernel threads

Kernel threads have emerged from the need to run kernel code in process context. Kernel threads are the basis of the workqueue mechanism. Essentially, a kernel thread is a thread that only runs in kernel mode and has no user address space or other user attributes.

To create a kernel thread, use `kthread_create()`:

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

structure task_struct *kthread_create(int (*threadfn)(void *data),
                                     void *data, const char namefmt[], ...);
```

- `threadfn` is a function that will be run by the kernel thread.
- `data` is a parameter to be sent to the function.

- `namefmt` represents the kernel thread name, as it is displayed in `ps/top`.

For example, the following call will create a kernel thread with the name `mykthread0`:

```
kthread_create(f, NULL, "%skthread%d", "my", 0);
```

To start the kernel thread, call `wake_up_process()`:

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

int wake_up_process(struct task_struct *p);
```

Alternatively, you can use `kthread_run()` to create and run a kernel thread:

```
struct task_struct *kthread_run(int (*threadfn)(void *data),
                                void *data, const char namefmt[], ...);
```

To stop a thread, use the `kthread_stop()` function. This function works by sending a signal to the thread. As a result, the thread function will not be interrupted in the middle of some important task. But, if the thread function never returns and does not check for signals, it will never actually stop.

LAB 7.3: "keyled class" module

In this lab, you will work with many of the concepts learned during the previous chapters, as well as those in the current chapter, to develop the driver. You will create a new class called `Keyled`. You will create several LED devices under the `Keyled` class and also several `sysfs` entries under each LED device. You will control each LED device by writing from user space to the `sysfs` entries under each LED device. In this driver, you will not initialize a `cdev` structure for each device (adding as an argument a `file_operations` structure) and add it to kernel space, so you will not see LED devices under `/dev` that you can control using system calls. You will control the LED devices by writing to the `sysfs` entries under `/sys/class/Keyled/<led_device>/` directory.

The period blinking value of each led device will be incremented or decremented via interrupts by using two buttons. A kernel thread will manage the led blinking, toggling the output value of the GPIO connected to the LED.

LAB 7.3 hardware description

You will use three pins of the Raspberry Pi to control the LEDs. These pins must be multiplexed as GPIOs in the DT. You will use the Raspberry Pi GPIO expansion connector to obtain the GPIOs.

To obtain the LEDs, you will use the Color click™ accessory board with mikroBUS™ form factor. See the Color click™ accessory board at <https://www.mikroe.com/color-click>. You can download the schematics from the above link or from the GitHub repository of this book.

Connect the GPIO27 pin to the Color click™ RD pin, GPIO22 pin to GR, and GPIO26 to BL.

To generate both interrupts, you will use the buttons of two MikroElektronika Button R click boards. See the board at <https://www.mikroe.com/button-r-click>. You can download the schematics from the above link or from the GitHub repository of this book. Connect the GPIO23 and GPIO24 pins to the INT pin of each Button R click board.

LAB 7.3 Device Tree description

You have to configure the pins GPIO27, GPIO22, GPIO26, GPIO23 and GPIO24 as GPIO signals.

Modify the bcm2710-rpi-3-b.dts Device Tree file by adding the following code in bold:

```
&gpio {
    spi0_pins: spi0_pins {
        brcm,pins = <9 10 11>;
        brcm,function = <4>; /* alt0 */
    };

    [...]

    key_pins: key_pins {
        brcm,pins = <23 24>;
        brcm,function = <0>;      /* Input */
        brcm,pull = <1 1>;      /* Pull down */
    };

    led_pins: led_pins {
        brcm,pins = <27 22 26>;
        brcm,function = <1>;      /* Output */
        brcm,pull = <1 1 1>;      /* Pull down */
    };
};

&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
    };
};
```

```

        status = "okay";
    };

    [...]

    ledpwm {
        compatible = "arrow,ledpwm";
        pinctrl-names = "default";
        pinctrl-0 = <&key_pins &led_pins>;

        bp1 {
            label = "MIKROBUS_KEY_1";
            gpios = <&gpio 23 GPIO_ACTIVE_LOW>;
            trigger = "falling";
        };

        bp2 {
            label = "MIKROBUS_KEY_2";
            gpios = <&gpio 24 GPIO_ACTIVE_LOW>;
            trigger = "falling";
        };

        ledred {
            label = "led";
            colour = "red";
            gpios = <&gpio 27 GPIO_ACTIVE_LOW>;
        };

        ledgreen {
            label = "led";
            colour = "green";
            gpios = <&gpio 22 GPIO_ACTIVE_LOW>;
        };

        ledblue {
            label = "led";
            colour = "blue";
            gpios = <&gpio 26 GPIO_ACTIVE_LOW>;
        };
    };

    [...]
};

```

LAB 7.3 code description of the "keyled class" module

The main code sections of the driver will now be described:

1. Include the function headers:

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

```
#include <linux/platform_device.h>
#include <linux/interrupt.h>
#include <linux/property.h>
#include <linux/kthread.h>
#include <linux/gpio/consumer.h>
#include <linux/delay.h>
#include <linux/spinlock.h>
```

2. Create a private structure that will store the specific data for each of the three led devices. The first field holds the name of each device. The `ledd` field holds the gpio descriptor of each pin connected to one of the three LEDs. The last field is a pointer to a `keyled_priv` structure that will hold the global data used for all the led devices. The `keyled_priv` structure will be analyzed in the next point.

```
struct led_device {
    char name[LED_NAME_LEN];
    struct gpio_desc *ledd;
    struct device *dev;
    struct keyled_priv *private;
};
```

3. Create a `keyled_priv` structure that will store global data accessible to all the LED devices. The first field of the private structure is the `num_leds` variable, which will store the number of LED devices declared in the DT. The `led_flag` field will tell you if there is any LED ON, to switch first all the LEDs OFF, before switching a new LED ON. The `task_flag` field will inform you if there is a kthread running. The `period` field holds the blinking period. The `period_lock` is a spinlock that will protect the access to the shared period variable between the user and interrupt context tasks. The task pointer variable will point to the `task_struct` structure returned by the `kthread_run()` function. The `led_class` field will point to the class structure returned by the `class_create()` function; this class structure will be used in calls to `device_create()`. The `dev` field holds your platform device. The `led_devt` field holds the first device identifier returned by the `alloc_chrdev_region()` function. The last field is an array of pointers pointing to each of the `led_device` private structures.

```
struct keyled_priv {
    u32 num_leds;
    u8 led_flag;
    u8 task_flag;
    u32 period;
    spinlock_t period_lock;
    struct task_struct *task;
    struct class *led_class;
    struct device *dev;
    dev_t led_devt;
    struct led_device *leds[];
};
```

4. These are the main points to set up the driver within the probe() function:
 - Declare a pointer to a fwnode_handle structure (struct fwnode_handle *child) and a pointer to the global private structure (struct keyed_priv *priv).
 - Get the number of LEDs and interrupt devices by using the device_get_child_node_count() function. You should get five devices returned.
 - Allocate the private structures by calling devm_kzalloc(). You will allocate space for the global structure and three pointers to led_device structures (see the sizeof_keyed_priv() function).
 - Allocate three device numbers with alloc_chrdev_region(), and create the keyed class with class_create().
 - Initialize a spinlock by using the spin_lock_init() function. The spinlock will be used to protect the access to the period variable. You will use spin_lock_irqsave() in user context and spin_lock() inside the ISR when using SMP architectures (for example, the BCM2837 SoC). For uniprocessor architectures, it is not needed to call spin_lock() inside the ISR, as the ISR cannot be executed in a different core to the one that has acquired the spinlock in user context.
 - The device_for_each_child_of_node() function walks for each child node, creating a sysfs device entry (under /sys/class/keyled/) for each found LED device by calling the led_device_register() function. You will get the GPIO descriptor of each GPIO declared inside each led node by using the devm_fwnode_get_gpiod_from_child() function, then the direction of the GPIO is set to output by calling gpiod_direction_output(). The GPIO descriptor of each GPIO declared inside each bp node is obtained by using the devm_fwnode_get_gpiod_from_child() function, then the direction of the GPIO is set to input by calling gpiod_direction_input(). The Linux IRQ numbers are obtained by using gpiod_to_irq(), and both interrupts are allocated by using devm_request_irq().

```
static int __init my_probe(struct platform_device *pdev)
{
    int count, ret, i;
    unsigned int major;
    struct fwnode_handle *child;
    struct device *dev = &pdev->dev;
    struct keyed_priv *priv;

    count = device_get_child_node_count(dev);

    priv = devm_kzalloc(dev, sizeof_keyed_priv(count-INT_NUMBER), GFP_KERNEL);

    /* Allocate 3 device numbers */
    alloc_chrdev_region(&priv->led_devt, 0, count-INT_NUMBER, "Keyled_class");
    major = MAJOR(priv->led_devt);
    dev_info(dev, "the major number is %d\n", major);
}
```

```

priv->led_class = class_create(THIS_MODULE, "keyled");

/* Create sysfs group */
priv->led_class->dev_groups = led_groups;
priv->dev = dev;

device_for_each_child_node(dev, child) {
    int irq, flags;
    struct gpio_desc *keyd;
    const char *label_name, *colour_name, *trigger;
    struct led_device *new_led;

    fwnode_property_read_string(child, "label", &label_name);

    if (strcmp(label_name, "led") == 0) {

        fwnode_property_read_string(child, "colour", &colour_name);
        /*
         * Create led devices under keyled class.
         * priv->num_leds is 0 for the first iteration.
         * The minor number for each device starts with 0
         * and is increased to the end of each iteration.
         */
        new_led = led_device_register(colour_name,
                                      priv->num_leds,
                                      dev,
                                      priv->led_devt,
                                      priv->led_class);

        new_led->ledd = devm_fwnode_get_gpiod_from_child(dev, NULL,
                                                         child,
                                                         GPIOD_ASIS,
                                                         colour_name);

        /* Associate each led struct with the global one */
        new_led->private = priv;

        /*
         * Point to each led struct
         * inside the global struct array of pointers
         */
        priv->leds[priv->num_leds] = new_led;
        priv->num_leds++;

        /* Set direction to output */
        gpiod_direction_output(new_led->ledd, 1);
        gpiod_set_value(new_led->ledd, 1);
    }
    else if (strcmp(label_name, "MIKROBUS_KEY_1") == 0) {

        keyd = devm_fwnode_get_gpiod_from_child(dev, NULL, child,
                                                GPIOD_ASIS,
                                                label_name);

        gpiod_direction_input(keyd);
    }
}

```

```

        fwnode_property_read_string(child, "trigger", &trigger);
        if (strcmp(trigger, "falling") == 0)
            flags = IRQF_TRIGGER_FALLING;
        else if (strcmp(trigger, "rising") == 0)
            flags = IRQF_TRIGGER_RISING;
        else if (strcmp(trigger, "both") == 0)
            flags = IRQF_TRIGGER_RISING |
                    IRQF_TRIGGER_FALLING;
        else
            return -EINVAL;
        irq = gpiod_to_irq(keyd);

        ret = devm_request_irq(dev, irq, KEY_ISR1,
                               flags, "ISR1", priv);
    }
    else if (strcmp(label_name, " MIKROBUS_KEY_2") == 0) {

        keyd = devm_fwnode_get_gpiod_from_child(dev, NULL, child,
                                                GPIOD_ASIS,
                                                label_name);

        gpiod_direction_input(keyd);
        fwnode_property_read_string(child, "trigger", &trigger);
        if (strcmp(trigger, "falling") == 0)
            flags = IRQF_TRIGGER_FALLING;
        else if (strcmp(trigger, "rising") == 0)
            flags = IRQF_TRIGGER_RISING;
        else if (strcmp(trigger, "both") == 0)
            flags = IRQF_TRIGGER_RISING |
                    IRQF_TRIGGER_FALLING;
        else
            return -EINVAL;

        irq = gpiod_to_irq(keyd);

        ret = devm_request_irq(dev, irq, KEY_ISR2,
                               flags, "ISR2", priv);
    }
    else {
        dev_info(dev, "Bad device tree value\n");
        ret = -EINVAL;
        goto error;
    }
}

/* Reset period to 10 */
priv->period = 10;

platform_set_drvdata(pdev, priv);
return 0;
}

```


5. In the `probe()` function, you will set a group of "sysfs attribute files" (to control each LED) with the line of code `priv->led_class->dev_groups = led_groups`. You have to declare outside of the `probe()` function the following structures:

```
static struct attribute *led_attrs[] = {
    &dev_attr_set_led.attr,
    &dev_attr_blink_on_led.attr,
    &dev_attr_blink_off_led.attr,
    &dev_attr_set_period.attr,
    NULL,
};

static const struct attribute_group led_group = {
    .attrs = led_attrs,
};

static const struct attribute_group *led_groups[] = {
    &led_group,
    NULL,
};
```

6. Write the sysfs functions which are called every time you write from user space (`/sys/class/Keyled/<led_device>/<attribute>`) to one of the next attributes (`set_led`, `blink_on`, `blink_off` and `set_period`). See below a brief description of what each function does:
- The `set_led_store()` function will receive two parameters ("on" and "off") from user space. Each `led_device` structure is recovered by using the `dev_get_drvdata()` function. In the `led_device_register()` function, called within the `probe()` function, was previously done the setting between each led device and its `led_device` structure by using the `dev_set_drvdata()` function. If there is a kthread running, then it is stopped. If the parameter received is "on", you will switch ON the specific LED by previously switching OFF all the LEDs. You will use `gpiod_set_value()` to perform this task. If the parameter received is "off", you will switch OFF the specific LED. The `led_flag` variable will be always set since the moment you switch ON the first LED, although all the LEDs are switched OFF later (I leave you as a task to modify the operation of this variable so that it is only set when any of the LEDs is ON during the execution of the driver).
 - The `blink_on_led_store()` function will receive an "on" parameter from user space. First of all, all the LEDs will be switched OFF, then if there is no any kthread running, it will be started a new one that will blink one of the LEDs with a specific period. If there is already a kthread running, the function will be exited.
 - The `blink_off_led_store()` function will receive an "off" parameter from user space. If there is a kthread running (blinking any of the LEDs), it will be stopped.
 - The `set_period_store()` function will set a new blinking period.

7. Write the two interrupt handlers. In this driver, an interrupt will be generated and handled each time you press one of the two buttons. In the handler, you will recover the global private structure from the ISR data argument. In one of the ISRs, the period variable will be increased by ten and in the other ISR decreased by the same value. The new value will be stored in the period variable of the global private structure. See below the ISR that increases the period:

```
static irqreturn_t KEY_ISR1(int irq, void *data)
{
    struct keyed_priv *priv = data;
    priv->period = priv->period + 10;
    if ((priv->period < 10) || (priv->period > 10000))
        priv->period = 10;
    return IRQ_HANDLED;
}
```

8. Write the thread function. Inside this function, you will recover the `led_device` structure that was set as a parameter in the `kthread_run()` function. The function `kthread_should_stop()` returns non-zero value if there is a stop request submitted by the `kthread_stop()` function. Until the call is exited, it will blink the specific LED by using the `gpio_set_value()` and `msleep()` functions.

```
static int led_flash(void *data) {
    u32 value = 0;
    struct led_device *led_dev = data;
    while(!kthread_should_stop()) {
        u32 period = led_dev->private->period;
        value = !value;
        gpio_set_value(led_dev->ledd, value);
        msleep(period/2);
    }
    gpio_set_value(led_dev->ledd, 1); /* switch off the led */
    dev_info(led_dev->dev, "Task completed\n");
    return 0;
};
```

9. Declare a list of devices supported by the driver:

```
static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,ledpwm" },
    {},
};

MODULE_DEVICE_TABLE(of, my_of_ids);
```

10. Add a `platform_driver` structure that will be registered to the platform bus:

```
static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "ledpwm",
```

```

        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    };
};

```

11. Register your driver with the platform bus:

```
module_platform_driver(my_platform_driver);
```

12. Create a new keyed_rpi3_class.c file in the linux_5.4_rpi3_drivers folder, and add keyed_rpi3_class.o to your Makefile obj-m variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

13. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

14. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 7-3: keyed_rpi3_class.c

```

#include <linux/fs.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/interrupt.h>
#include <linux/property.h>
#include <linux/kthread.h>
#include <linux/gpio/consumer.h>
#include <linux/delay.h>
#include <linux/spinlock.h>
#include <linux/of_device.h>

#define LED_NAME_LEN      32
#define INT_NUMBER        2
static const char *HELLO_KEYS_NAME1 = "MIKROBUS_KEY1";
static const char *HELLO_KEYS_NAME2 = "MIKROBUS_KEY2";

/* Specific LED private structure */
struct led_device {
    char name[LED_NAME_LEN];
    struct gpio_desc *ledd; /* each LED gpio_desc */
    struct device *dev;
    struct keyed_priv *private; /* pointer to the global private struct */
};

/* Global private structure */
struct keyed_priv {

```

```

u32 num_leds;
u8 led_flag;
u8 task_flag;
u32 period;
spinlock_t period_lock;
struct task_struct *task;          /* kthread task_struct */
struct class *led_class;          /* the keyed class */
struct device *dev;
dev_t led_devt;                   /* first device identifier */
struct led_device *leds[];        /* pointers to each led private struct */
};

/* kthread function */
static int led_flash(void *data){
    unsigned long flags;
    u32 value = 0;
    struct led_device *led_dev = data;
    dev_info(led_dev->dev, "Task started\n");
    dev_info(led_dev->dev, "I am inside the kthread\n");
    while(!kthread_should_stop()) {
        spin_lock_irqsave(&led_dev->private->period_lock, flags);
        u32 period = led_dev->private->period;
        spin_unlock_irqrestore(&led_dev->private->period_lock, flags);
        value = !value;
        gpiod_set_value(led_dev->ledd, value);
        msleep(period/2);
    }
    gpiod_set_value(led_dev->ledd, 1); /* switch off the led */
    dev_info(led_dev->dev, "Task completed\n");
    return 0;
};

/*
 * Sysfs methods
 */

/* Switch on/of each led */
static ssize_t set_led_store(struct device *dev,
                           struct device_attribute *attr,
                           const char *buf, size_t count)
{
    int i;
    char *buffer = buf;
    struct led_device *led_count;
    struct led_device *led = dev_get_drvdata(dev);

    /* Replace \n added from terminal with \0 */
    *(buffer+(count-1)) = '\0';

    if (led->private->task_flag == 1) {
        kthread_stop(led->private->task);
        led->private->task_flag = 0;
    }
}

```

```

if(!strcmp(buffer, "on")) {
    if (led->private->led_flag == 1) {
        for (i = 0; i < led->private->num_leds; i++) {
            led_count = led->private->leds[i];
            gpiod_set_value(led_count->ledd, 1);
        }
        gpiod_set_value(led->ledd, 0);
    }
    else {
        gpiod_set_value(led->ledd, 0);
        led->private->led_flag = 1;
    }
}
else if (!strcmp(buffer, "off")) {
    gpiod_set_value(led->ledd, 1);
}
else {
    dev_info(led->dev, "Bad led value.\n");
    return -EINVAL;
}

return count;
}
static DEVICE_ATTR_WO(set_led);

/* Blink an LED by running a kthread */
static ssize_t blink_on_led_store(struct device *dev,
                                struct device_attribute *attr,
                                const char *buf, size_t count)
{
    int i;
    char *buffer = buf;
    struct led_device *led_count;
    struct led_device *led = dev_get_drvdata(dev);

    /* Replace \n added from terminal with \0 */
    *(buffer+(count-1)) = '\0';

    if (led->private->led_flag == 1) {
        for (i = 0; i < led->private->num_leds; i++) {
            led_count = led->private->leds[i];
            gpiod_set_value(led_count->ledd, 1);
        }
    }

    if(!strcmp(buffer, "on")) {
        if (led->private->task_flag == 0)
        {
            led->private->task = kthread_run(led_flash, led, "Led_flash_tread");
            if(IS_ERR(led->private->task)) {
                dev_info(led->dev, "Failed to create the task\n");
                return PTR_ERR(led->private->task);
            }
        }
    }
}

```

```

        else
            return -EBUSY;
    }
    else {
        dev_info(led->dev, "Bad led value.\n");
        return -EINVAL;
    }

    led->private->task_flag = 1;

    dev_info(led->dev, "Blink_on_led exited\n");
    return count;
}
static DEVICE_ATTR_WO(blink_on_led);

/* Switch off LED blinking */
static ssize_t blink_off_led_store(struct device *dev,
                                   struct device_attribute *attr,
                                   const char *buf, size_t count)
{
    int i;
    char *buffer = buf;
    struct led_device *led = dev_get_drvdata(dev);
    struct led_device *led_count;

    /* Replace \n added from terminal with \0 */
    *(buffer+(count-1)) = '\0';

    if(!strcmp(buffer, "off")) {
        if (led->private->task_flag == 1) {
            kthread_stop(led->private->task);
            for (i = 0; i < led->private->num_leds; i++) {
                led_count = led->private->leds[i];
                gpiod_set_value(led_count->ledd, 1);
            }
        }
        else
            return 0;
    }
    else {
        dev_info(led->dev, "Bad led value.\n");
        return -EINVAL;
    }

    led->private->task_flag = 0;
    return count;
}
static DEVICE_ATTR_WO(blink_off_led);

/* Set the blinking period */
static ssize_t set_period_store(struct device *dev,
                                struct device_attribute *attr,
                                const char *buf, size_t count)
{

```

```

    unsigned long flags;
    int ret, period;
    struct led_device *led = dev_get_drvdata(dev);
    dev_info(led->dev, "Enter set_period\n");

    ret = sscanf(buf, "%u", &period);
    if (ret < 1 || period < 10 || period > 10000) {
        dev_err(dev, "invalid value\n");
        return -EINVAL;
    }

    spin_lock_irqsave(&led->private->period_lock, flags);
    led->private->period = period;
    spin_unlock_irqrestore(&led->private->period_lock, flags);

    dev_info(led->dev, "period is set\n");
    return count;
}
static DEVICE_ATTR_WO(set_period);

/* Declare the sysfs structures */
static struct attribute *led_attrs[] = {
    &dev_attr_set_led.attr,
    &dev_attr_blink_on_led.attr,
    &dev_attr_blink_off_led.attr,
    &dev_attr_set_period.attr,
    NULL,
};

static const struct attribute_group led_group = {
    .attrs = led_attrs,
};

static const struct attribute_group *led_groups[] = {
    &led_group,
    NULL,
};

/*
 * Allocate space for the global private struct
 * and the three LED private structs
 */
static inline int sizeof_keyled_priv(int num_leds)
{
    return sizeof(struct keyled_priv) + (sizeof(struct led_device*) * num_leds);
}

/* First interrupt handler */
static irqreturn_t KEY_ISR1(int irq, void *data)
{
    struct keyled_priv *priv = data;
    dev_info(priv->dev, "interrupt MIKROBUS_KEY1 received. key: %s\n", HELLO_KEYS_NAME1);

    spin_lock(&priv->period_lock);

```

```

priv->period = priv->period + 10;
if ((priv->period < 10) || (priv->period > 10000))
    priv->period = 10;
spin_unlock(&priv->period_lock);

dev_info(priv->dev, "the led period is %d\n", priv->period);
return IRQ_HANDLED;
}

/* Second interrupt handler */
static irqreturn_t KEY_ISR2(int irq, void *data)
{
    struct keyed_priv *priv = data;
    dev_info(priv->dev, "interrupt MIKROBUS_KEY2 received. key: %s\n", HELLO_KEYS_NAME2);

    spin_lock(&priv->period_lock);
    priv->period = priv->period - 10;
    if ((priv->period < 10) || (priv->period > 10000))
        priv->period = 10;
    spin_unlock(&priv->period_lock);

    dev_info(priv->dev, "the led period is %d\n", priv->period);
    return IRQ_HANDLED;
}

/* Create the LED devices under the sysfs keyed entry */
struct led_device *led_device_register(const char *name, int count,
                                       struct device *parent, dev_t led_devt,
                                       struct class *led_class)
{
    struct led_device *led;
    dev_t devt;
    int ret;

    /* Allocate a new led device */
    led = devm_kzalloc(parent, sizeof(struct led_device), GFP_KERNEL);
    if (!led)
        return ERR_PTR(-ENOMEM);

    /* Get the minor number of each device */
    devt = MKDEV(MAJOR(led_devt), count);

    /* Create the device and init the device data */
    led->dev = device_create(led_class, parent, devt, led, "%s", name);
    if (IS_ERR(led->dev)) {
        dev_err(led->dev, "unable to create device %s\n", name);
        ret = PTR_ERR(led->dev);
        return ERR_PTR(ret);
    }

    dev_info(led->dev, "the major number is %d\n", MAJOR(led_devt));
    dev_info(led->dev, "the minor number is %d\n", MINOR(devt));

    /* To recover later from each sysfs entry */

```



```

dev_set_drvdata(led->dev, led);

strncpy(led->name, name, LED_NAME_LEN);

dev_info(led->dev, "led %s added\n", led->name);

return led;
}

static int my_probe(struct platform_device *pdev)
{
    int count, ret, i;
    unsigned int major;
    struct fwnode_handle *child;

    struct device *dev = &pdev->dev;
    struct keyled_priv *priv;

    dev_info(dev, "my_probe() function is called.\n");

    count = device_get_child_node_count(dev);
    if (!count)
        return -ENODEV;

    dev_info(dev, "there are %d nodes\n", count);

    /* Allocate all the private structures */
    priv = devm_kzalloc(dev, sizeof_keyled_priv(count-INT_NUMBER), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;

    /* Allocate 3 device numbers */
    alloc_chrdev_region(&priv->led_devt, 0, count-INT_NUMBER, "Keyled_class");
    major = MAJOR(priv->led_devt);
    dev_info(dev, "the major number is %d\n", major);

    /* Create the LED class */
    priv->led_class = class_create(THIS_MODULE, "keyled");
    if (!priv->led_class) {
        dev_info(dev, "failed to allocate class\n");
        return -ENOMEM;
    }

    /* Set attributes for the LED class */
    priv->led_class->dev_groups = led_groups;
    priv->dev = dev;

    spin_lock_init(&priv->period_lock);

    /* Parse all the DT nodes */
    device_for_each_child_node(dev, child) {
        int irq, flags;
        struct gpio_desc *keyd;
        const char *label_name, *colour_name, *trigger;

```

```

struct led_device *new_led;

fwnode_property_read_string(child, "label", &label_name);

/* Parsing the DT LED nodes */
if (strcmp(label_name, "led") == 0) {

    fwnode_property_read_string(child, "colour", &colour_name);

    /*
     * Create led devices under keyed class
     * priv->num_leds is 0 for the first iteration
     * The minor number for each device starts with 0
     * and is increased to the end of each iteration
     */
    new_led = led_device_register(colour_name, priv->num_leds, dev,
                                  priv->led_devt, priv->led_class);
    if (!new_led) {

        fwnode_handle_put(child);
        ret = PTR_ERR(new_led);

        for (i = 0; i < priv->num_leds-1; i++) {
            device_destroy(priv->led_class,
                           MKDEV(MAJOR(priv->led_devt), i));
        }
        class_destroy(priv->led_class);
        return ret;
    }

    new_led->ledd = devm_fwnode_get_gpiod_from_child(dev, NULL, child,
                                                    GPIOD_ASIS,
                                                    colour_name);

    if (IS_ERR(new_led->ledd)) {
        fwnode_handle_put(child);
        ret = PTR_ERR(new_led->ledd);
        goto error;
    }
    new_led->private = priv;
    priv->leds[priv->num_leds] = new_led;
    priv->num_leds++;

    /* Set direction to output */
    gpiod_direction_output(new_led->ledd, 1);

    /* set led state to off */
    gpiod_set_value(new_led->ledd, 1);
}

/* Parsing the interrupt nodes */
else if (strcmp(label_name, "MIKROBUS_KEY_1") == 0) {
    keyd = devm_fwnode_get_gpiod_from_child(dev, NULL, child,
                                             GPIOD_ASIS, label_name);
    gpiod_direction_input(keyd);
}

```

```

    fwnode_property_read_string(child, "trigger", &trigger);
    if (strcmp(trigger, "falling") == 0)
        flags = IRQF_TRIGGER_FALLING;
    else if (strcmp(trigger, "rising") == 0)
        flags = IRQF_TRIGGER_RISING;
    else if (strcmp(trigger, "both") == 0)
        flags = IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING;
    else
        return -EINVAL;

    irq = gpiod_to_irq(keyd);
    if (irq < 0)
        return irq;

    ret = devm_request_irq(dev, irq, KEY_ISR1, flags, "ISR1", priv);
    if (ret) {
        dev_err(dev, "Failed to request interrupt %d, error %d\n",
            irq, ret);
        return ret;
    }
    dev_info(dev, "IRQ number: %d\n", irq);
}
else if (strcmp(label_name, "MIKROBUS_KEY_2") == 0) {

    keyd = devm_fwnode_get_gpiod_from_child(dev, NULL, child,
                                            GPIOD_ASIS, label_name);

    gpiod_direction_input(keyd);
    fwnode_property_read_string(child, "trigger", &trigger);
    if (strcmp(trigger, "falling") == 0)
        flags = IRQF_TRIGGER_FALLING;
    else if (strcmp(trigger, "rising") == 0)
        flags = IRQF_TRIGGER_RISING;
    else if (strcmp(trigger, "both") == 0)
        flags = IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING;
    else
        return -EINVAL;

    irq = gpiod_to_irq(keyd);
    if (irq < 0)
        return irq;

    ret = devm_request_irq(dev, irq, KEY_ISR2, flags, "ISR2", priv);
    if (ret < 0) {
        dev_err(dev, "Failed to request interrupt %d, error %d\n",
            irq, ret);
        goto error;
    }
    dev_info(dev, "IRQ number: %d\n", irq);
}
else {
    dev_info(dev, "Bad device tree value\n");
    ret = -EINVAL;
    goto error;
}
}

```

```

    }

    dev_info(dev, "i am out of the device tree\n");

    /* Reset period to 10 */
    priv->period = 10;

    dev_info(dev, "the led period is %d\n", priv->period);

    platform_set_drvdata(pdev, priv);

    dev_info(dev, "my_probe() function is exited.\n");

    return 0;

error:
    /* Unregister everything in case of errors */
    for (i = 0; i < priv->num_leds; i++) {
        device_destroy(priv->led_class, MKDEV(MAJOR(priv->led_devt), i));
    }
    class_destroy(priv->led_class);
    unregister_chrdev_region(priv->led_devt, priv->num_leds);
    return ret;
}

static int my_remove(struct platform_device *pdev)
{
    int i;
    struct led_device *led_count;
    struct keyed_priv *priv = platform_get_drvdata(pdev);
    dev_info(&pdev->dev, "my_remove() function is called.\n");

    if (priv->task_flag == 1) {
        kthread_stop(priv->task);
        priv->task_flag = 0;
    }

    if (priv->led_flag == 1) {
        for (i = 0; i < priv->num_leds; i++) {
            led_count = priv->leds[i];
            gpio_set_value(led_count->ledd, 1);
        }
    }

    for (i = 0; i < priv->num_leds; i++) {
        device_destroy(priv->led_class, MKDEV(MAJOR(priv->led_devt), i));
    }
    class_destroy(priv->led_class);
    unregister_chrdev_region(priv->led_devt, priv->num_leds);
    dev_info(&pdev->dev, "my_remove() function is exited.\n");
    return 0;
}

static const struct of_device_id my_of_ids[] = {

```

```

    { .compatible = "arrow,ledpwm"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "ledpwm",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};
module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a platform keyed_class driver that decreases \
    and increases the LED flashing period");

```

keyled_rpi3_class.ko demonstration

Load the module:

```

root@raspberrypi:/home/pi# insmod keyled_rpi3_class.ko
ledpwm soc:ledpwm: my_probe() function is called.
ledpwm soc:ledpwm: there are 5 nodes
ledpwm soc:ledpwm: the major number is 238
ledpwm soc:ledpwm: IRQ number: 166
ledpwm soc:ledpwm: IRQ number: 167
keyled red: the major number is 238
keyled red: the minor number is 0
keyled red: led red added
keyled green: the major number is 238
keyled green: the minor number is 1
keyled green: led green added
keyled blue: the major number is 238
keyled blue: the minor number is 2
keyled blue: led blue added
ledpwm soc:ledpwm: i am out of the device tree
ledpwm soc:ledpwm: the led period is 10
ledpwm soc:ledpwm: my_probe() function is exited.

```

Check the interrupts of the Raspberry Pi. See the linux IRQ numbers (166 and 167) with their hwirq numbers (23 and 24) for the pinctrl-bcm2835 controller:

```

root@raspberrypi:/home/pi# cat /proc/interrupts

```

	CPU0	CPU1	CPU2	CPU3			
17:	225	0	0	0	ARMCTRL-level	1 Edge	3f
00b880.mailbox							
18:	715	0	0	0	ARMCTRL-level	2 Edge	VC
HIQ doorbell							

```

40:          0          0          0          0 ARMCTRL-level 48 Edge    bc
m2708_fb DMA
42:        215          0          0          0 ARMCTRL-level 50 Edge    DM
A IRQ
44:        5948         0          0          0 ARMCTRL-level 52 Edge    DM

[...]

166:         0          0          0          0 pinctrl-bcm2835 23 Edge
ISR1
167:         0          0          0          0 pinctrl-bcm2835 24 Edge
ISR2
FIQ:                usb_fiq
IPI0:         0          0          0          0 CPU wakeup interrupts
IPI1:         0          0          0          0 Timer broadcast interrupts
IPI2:        3277        3614        7643        3359 Rescheduling interrupts
IPI3:         374         655        1273        1428 Function call interrupts
IPI4:         0          0          0          0 CPU stop interrupts
IPI5:         71         86         165         33  IRQ work interrupts
IPI6:         0          0          0          0 completion interrupts
Err:          0

```

See the devices under the keyed class:

```

root@raspberrypi:/home/pi# ls /sys/class/keyled/
blue green red

```

Switch ON and OFF the led devices:

```

root@raspberrypi:/home/pi# echo on > /sys/class/keyled/red/set_led
root@raspberrypi:/home/pi# echo on > /sys/class/keyled/blue/set_led
root@raspberrypi:/home/pi# echo on > /sys/class/keyled/green/set_led

```

Switch OFF the green LED:

```

root@raspberrypi:/home/pi# echo off > /sys/class/keyled/green/set_led

```

Blink the green LED:

```

root@raspberrypi:/home/pi# echo on > /sys/class/keyled/green/blink_on_led
keyled green: Blink_on_led exited
keyled green: Task started
keyled green: I am inside the kthread

```

Stop blinking the green LED:

```

root@raspberrypi:/home/pi# echo off > /sys/class/keyled/green/blink_off_led
keyled green: Task completed

```

Blink the red LED:

```

root@raspberrypi:/home/pi# echo on > /sys/class/keyled/red/blink_on_led
keyled red: Blink_on_led exited
keyled red: Task started
keyled red: I am inside the kthread

```

Change the blinking period for the red LED:

```
root@raspberrypi:/home/pi# echo 100 > /sys/class/keyled/red/set_period
keyled red: Enter set_period
keyled red: period is set
```

Increase the blinking period pressing the MIKROBUS_KEY1 button, and decrease the blinking period pressing the MIKROBUS_KEY2 button:

```
root@raspberrypi:/home/pi#
ledpwm soc:ledpwm: interrupt MIKROBUS_KEY2 received. key: MIKROBUS_KEY2
ledpwm soc:ledpwm: the led period is 90
ledpwm soc:ledpwm: interrupt MIKROBUS_KEY2 received. key: MIKROBUS_KEY2
ledpwm soc:ledpwm: the led period is 80
ledpwm soc:ledpwm: interrupt MIKROBUS_KEY2 received. key: MIKROBUS_KEY2
ledpwm soc:ledpwm: the led period is 70
ledpwm soc:ledpwm: interrupt MIKROBUS_KEY2 received. key: MIKROBUS_KEY2
ledpwm soc:ledpwm: the led period is 60
ledpwm soc:ledpwm: interrupt MIKROBUS_KEY1 received. key: MIKROBUS_KEY1
ledpwm soc:ledpwm: the led period is 70
ledpwm soc:ledpwm: interrupt MIKROBUS_KEY1 received. key: MIKROBUS_KEY1
ledpwm soc:ledpwm: the led period is 80
ledpwm soc:ledpwm: interrupt MIKROBUS_KEY1 received. key: MIKROBUS_KEY1
ledpwm soc:ledpwm: the led period is 90
```

Exit with ^C:

```
root@raspberrypi:/home/pi#
```

Stop Blinking the red LED:

```
root@raspberrypi:/home/pi# echo off > /sys/class/keyled/red/blink_off_led
keyled red: Task completed
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod keyled_rpi3_class.ko
ledpwm ledpwm: my_remove() function is called.
ledpwm ledpwm: my_remove() function is exited
```

LAB 7.4: "GPIO expander device" module

This LAB 7.4 will help to reinforce the concepts of creating **NESTED THREADED GPIO irqchips** drivers, which were explained during this Chapter 7, and apply in a practical way how to create a gpio controller with interrupt capabilities. You will also develop a user application that requests GPIO interrupts from user space by using the GPIOlib APIs.

This new kernel module will control the Cypress CY8C9520A device. The CY8C9520A is a multi-port IO expander with on board user available EEPROM and several PWM outputs. The IO expander's data pins can be independently assigned as inputs, outputs, quasi-bidirectional input/outputs or PWM outputs. The individual data pins can be configured as open drain or collector, strong drive (10 mA source, 25 mA sink), resistively pulled up or down, or high impedance. The factory default configuration is pulled up internally. You can check all the info related to this device at <https://www.cypress.com/products/cy8c95xx>.

In this lab, you will use the EXPAND 6 Click from MIKROE. The documentation of this board can be found at <https://www.mikroe.com/expand-6-click>.

Not all the CY8C9520A features are included in this driver. The driver will configure the CY8C9520A port pins as input and outputs and handle GPIO interrupts.

LAB 7.4 hardware description

In this lab, you will use the I2C pins of the Raspberry Pi 40-pin GPIO header, which is found on all current Raspberry Pi boards, to connect to the EXPAND 6 Click mikroBUS™ socket. In the following image, you can see the EXPAND 6 Click mikroBUS™ socket:

Notes	Pin	 mikroBUS				Pin	Notes
	NC	1	AN	PWM	16	NC	
Reset	RST	2	RST	INT	15	INT	Interrupt
	NC	3	CS	RX	14	NC	
	NC	4	SCK	TX	13	NC	
	NC	5	MISO	SCL	12	SCL	I2C Clock
	NC	6	MOSI	SDA	11	SDA	I2C Data
Power Supply	3.3V	7	3.3V	5V	10	5V	Power Supply
Ground	GND	8	GND	GND	9	GND	Ground

Connect the Raspberry Pi’s I2C pins to the I2C ones of the CY8C9520A device (obtained from the EXPAND 6 Click mikroBUS™ socket):

- Connect Raspberry Pi **SCL** to CY8C9520A **SCL** (Pin 12 of Mikrobus).
- Connect Raspberry Pi **SDA** to CY8C9520A **SDA** (Pin 11 of Mikrobus).
- Connect Raspberry Pi **GPIO23** to CY8C9520A **INT** (Pin 15 of Mikrobus).

Connect the next power pins between the two boards:

- Connect Raspberry Pi **3.3V** to CY8C9520A **3.3V** (Pin 7 of Mikrobus).
- Connect Raspberry Pi **GND** to CY8C9520A **GND** (Pin 8 of Mikrobus).

The hardware setup between the two boards is already done!!

LAB 7.4 Device Tree description

Open the `bcm2710-rpi-3-b.dts` DT file and find the `i2c1` controller master node. Inside the `i2c1` node, you can see a `pinctrl-0` property that points to the `i2c1_pins` pin configuration node, which configures the pins of the `i2c1` controller in I2C mode. The `i2c1_pins` pin configuration node is defined in the `bcm2710-rpi-3-b.dts` file, inside the `gpio` node property.

The `i2c1` controller is enabled by writing "okay" to the status property. You will set to 100KHz the clock-frequency property. EXPAND 6 Click communicates with the Raspberry Pi using an I2C bus interface with a maximum frequency of 100kHz.

Now, you will add the `cy8c9520a` node to the `i2c1` controller node. There must be a DT device node's compatible property identical to the compatible string stored in one of the driver's of `_device_id` structures. The `reg` property includes the I2C address of the device.

The interrupt-controller property is an empty property which declares a node as a device that receives interrupt signals. The `interrupt-cells` property is a property of the interrupt controller that defines how many cells are needed to specify a single interrupt in an interrupt client node. In our device node, the `interrupt-cells` property is set to two. The first cell defines the index of the interrupt within the controller, while the second cell is used to specify the trigger and level flags of the interrupt.

Every GPIO controller node must contain both an empty `gpio-controller` property and a `gpio-cells` integer property, which indicates the number of cells in a `gpio-specifier` for a `gpio` client device.

The `interrupt-parent` is a property containing a phandle to the interrupt controller that it is attached to. Nodes that do not have an `interrupt-parent` property can also inherit the property of their parent node. The CY8C9520A interrupt pin (INT) is connected to the GPIO23 pin of the BCM2837 SoC, so the interrupt parent of our device is the `gpio` peripheral of the BCM2837 SoC.

The `interrupts` property is a property containing a list of interrupt specifiers, one for each interrupt output signal on the device. In our driver, there is one output interrupt, so only one interrupt specifier containing the interrupted line number of the `GPIO` peripheral is needed.

The Device Tree configuration of our `cy8c9520a` device is shown below:

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
    status = "okay";

    [...]

    cy8c9520a: cy8c9520a@20 {
        compatible = "cy8c9520a";
```

```

        reg = <0x20>;
        interrupt-controller;
        #interrupt-cells = <2>;
        gpio-controller;
        #gpio-cells = <2>;

        interrupts = <23 1>;
        interrupt-parent = <&gpio>;
    };

};

```

LAB 7.4 GPIO controller driver description

The main code sections of the driver will be described using two different categories: I2C driver setup and GPIO driver interface. Our CY8C9520A driver is based on the CY8C9540A driver from Intel Corporation.

I2C driver setup

These are the main code sections:

1. Include the function headers:

```
#include <linux/i2c.h>
```

2. Create an i2c_driver structure:

```

static struct i2c_driver cy8c9520a_driver = {
    .driver = {
        .name = DRV_NAME,
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    },
    .probe = cy8c9520a_probe,
    .remove = cy8c9520a_remove,
    .id_table = cy8c9520a_id,
};

```

3. Register to the I2C bus as a driver:

```
module_i2c_driver(cy8c9520a_driver);
```

4. Add "cy8c9520a" to the list of devices supported by the driver. The compatible variable matches with the compatible property of the cy8c9520a DT node:

```

static const struct of_device_id my_of_ids[] = {
    { .compatible = "cy8c9520a" },
    {}
};

MODULE_DEVICE_TABLE(of, my_of_ids);

```

5. Define an array of `i2c_device_id` structures:

```
static const struct i2c_device_id cy8c9520a_id[] = {
    {DRV_NAME, 0},
    {}
};
MODULE_DEVICE_TABLE(i2c, cy8c9520a_id);
```

GPIO driver interface

The CY8C9520A driver will control the I/O expander's data pins as inputs and outputs. In this driver, each and every GPIO pin can be used as an external interrupt. Whenever there is an input change on a specific GPIO pin, an interrupt output (INT) will be asserted by the CY8C9520A GPIO controller.

The CY8C9520A driver will register its `gpio_chip` structure with the kernel and its `irq_chip` structure with the IRQ subsystem.

Our GPIO `irqchip` will fall in the category of NESTED THREADED GPIO IRQCHIPS, which are off-chip GPIO expanders that reside on the other side of a sleeping bus, such as I2C or SPI.

The GPIOlib framework will provide the kernel and user space APIs to control the GPIOs and handle their interrupts.

These are the main steps to create our CY8C9520A driver, which includes a GPIO controller with interrupt capabilities:

1. Include the following header which declares the structures that will be used to create a GPIO driver:

```
#include <linux/gpio/driver.h>
```

2. Initialize the `gpio_chip` structure with the different callbacks that will control the gpio lines of the GPIO controller, and register the `gpiochip` with the kernel by using the `devm_gpiochip_add_data()` function. In the Listing 7-4, you can check the source code of these callback functions. Comments have been added before the main lines of the code to understand the meaning of the same:

```
static int cy8c9520a_gpio_init(struct cy8c9520a *cygpio)
{
    struct gpio_chip *gpiochip = &cygpio->gpio_chip;
    int err;

    gpiochip->label = cygpio->client->name;
    gpiochip->base = -1;
    gpiochip->ngpio = NGPIO;
    gpiochip->parent = &cygpio->client->dev;
    gpiochip->of_node = gpiochip->parent->of_node;
    gpiochip->can_sleep = true;
```

```

gpiochip->direction_input = cy8c9520a_gpio_direction_input;
gpiochip->direction_output = cy8c9520a_gpio_direction_output;
gpiochip->get = cy8c9520a_gpio_get;
gpiochip->set = cy8c9520a_gpio_set;
gpiochip->owner = THIS_MODULE;

/* register a gpio_chip */
err = devm_gpiochip_add_data(gpiochip->parent, gpiochip, cygpio);
if (err)
    return err;
return 0;
}

```

3. Initialize the `irq_chip` structure with the different callbacks that will handle the GPIO interrupts flow. In the Listing 7-4, you can check the source code of these callback functions. Comments have been added before the main lines of the code to understand the meaning of the same:

```

static struct irq_chip cy8c9520a_irq_chip = {
    .name                = "cy8c9520a-irq",
    .irq_mask             = cy8c9520a_irq_mask,
    .irq_unmask          = cy8c9520a_irq_unmask,
    .irq_bus_lock        = cy8c9520a_irq_bus_lock,
    .irq_bus_sync_unlock = cy8c9520a_irq_bus_sync_unlock,
    .irq_set_type        = cy8c9520a_irq_set_type,
};

```

4. Write the interrupt setup function (`cy8c9520a_irq_setup`). The `gpiochip_irqchip_add_nested()` function adds a nested irqchip to a gpiochip. This function takes as a parameter the `handle_simple_irq` flow handler, which handles simple interrupts sent from a demultiplexing interrupt handler or coming from hardware where no interrupt hardware control is necessary. You can find all the complete information about irq-flow methods at <https://www.kernel.org/doc/html/latest/core-api/genericirq.html>.

The interrupt handler for the GPIO child device will be called inside of a new thread created by the `handle_nested_irq()` function, which is called inside the interrupt handler of the CY8C9520A driver.

The `devm_request_threaded_irq()` function (called in the `cy8c9520a_irq_setup` function) will allocate the interrupt line, taking as parameters: the interrupt handler of the driver, the linux IRQ number (`client->irq`), flags (`IRQF_ONESHOT | IRQF_TRIGGER_HIGH`) that will instruct the kernel about the desired behaviour, and a pointer to the `cygpio` global structure, which will be recovered in the interrupt handler of the driver.

```

static int cy8c9520a_irq_setup(struct cy8c9520a *cygpio)
{
    struct i2c_client *client = cygpio->client;
    struct gpio_chip *chip = &cygpio->gpio_chip;
    u8 dummy[NPORTS];

```

```

int ret, i;

mutex_init(&cygpio->irq_lock);

/*
 * Clear interrupt state registers by reading the three registers
 * Interrupt Status Port0, Interrupt Status Port1,
 * Interrupt Status Port2,
 * and store the values in a dummy array
 */
i2c_smbus_read_i2c_block_data(client, REG_INTR_STAT_PORT0, NPORTS, dummy);

/*
 * Initialize Interrupt Mask Port Register (19h) for each port
 * Disable the activation of the INT lines. Each 1 in this
 * register, masks (disables) the INT for the corresponding GPIO
 */
memset(cygpio->irq_mask_cache, 0xff, sizeof(cygpio->irq_mask_cache));
memset(cygpio->irq_mask, 0xff, sizeof(cygpio->irq_mask));

/* Disable interrupts in all the gpio lines */
for (i = 0; i < NPORTS; i++) {
    i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
    i2c_smbus_write_byte_data(client, REG_INTR_MASK, cygpio->irq_mask[i]);
}

/* add a nested irqchip to the gpiochip */
gpiochip_irqchip_add_nested(chip,
                           &cy8c9520a_irq_chip,
                           0,
                           handle_simple_irq,
                           IRQ_TYPE_NONE);

/*
 * Request interrupt on a GPIO pin of the external processor
 * this processor pin is connected to the INT pin of the cy8c9520a
 */
devm_request_threaded_irq(&client->dev, client->irq, NULL,
                        cy8c9520a_irq_handler,
                        IRQF_ONESHOT | IRQF_TRIGGER_HIGH,
                        dev_name(&client->dev), cygpio);

/*
 * set up a nested irq handler
 * you can now request interrupts from GPIO child drivers nested
 * to the cy8c9520a driver
 */
gpiochip_set_nested_irqchip(chip, &cy8c9520a_irq_chip, client->irq);

return 0;
err:
mutex_destroy(&cygpio->irq_lock);
return ret;
}

```

5. Write the interrupt handler. Inside this handler, the pending GPIO interrupts are checked by reading the pending variable value, then the position of the first bit set in the variable is returned; the `_ffs()` function is used to perform this task. For each pending interrupt that is found, there is a call to the `handle_nested_irq()` wrapper function, which in turn calls the interrupt handler of the GPIO child driver that requested a GPIO interrupt by using the `devm_request_threaded_irq()` function. The parameter of the `handle_nested_irq()` function is the Linux IRQ number previously returned by using the `irq_find_mapping()` function, which receives the `hwirq` of the input pin as a parameter (`gpio_irq` variable). The pending interrupt is cleared by doing `pending &= ~BIT(gpio)`, and the same process is repeated until all the pending interrupts are being managed.

```
static irqreturn_t cy8c9520a_irq_handler(int irq, void *devid)
{
    struct cy8c9520a *cygpio = devid;
    u8 stat[NPORTS], pending;
    unsigned port, gpio, gpio_irq;
    int ret;

    /*
     * store in stat and clear (to enable ints)
     * the three interrupt status registers by reading them
     */
    i2c_smbus_read_i2c_block_data(cygpio->client, REG_INTR_STAT_PORT0, NPORTS, stat);

    ret = IRQ_NONE;

    for (port = 0; port < NPORTS; port++) {
        mutex_lock(&cygpio->irq_lock);

        /*
         * In every port, check the GPIOs that have their INT unmasked
         * and whose bits have been enabled in their REG_INTR_STAT_PORT
         * register due to an interrupt in the GPIO, then store the new
         * value in the pending register
         */
        pending = stat[port] & (~cygpio->irq_mask[port]);
        mutex_unlock(&cygpio->irq_lock);

        while (pending) {
            ret = IRQ_HANDLED;
            /* get the first gpio that has got an INT */
            gpio = __ffs(pending);

            /* clears the gpio in the pending register */
            pending &= ~BIT(gpio);

            /* gets the INT number associated to this gpio */
            gpio_irq = cy8c9520a_port_offs[port] + gpio;

            /* launch the ISR of the GPIO child driver */

```

```

        handle_nested_irq(irq_find_mapping(cygpio->gpio_chip.irq.domain,
        gpio_irq));
    }
}

return ret;
}

```

6. Create a new `linux_5.4_CY8C9520A_driver` folder inside the `linux_5.4_rpi3_drivers` folder:

```
~/linux_5.4_rpi3_drivers$ mkdir linux_5.4_CY8C9520A_driver
```

7. Create a new `CY8C9520A_rpi3.c` file and a Makefile file in the `linux_5.4_CY8C9520A_driver` folder, and add `CY8C9520A_rpi3.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver$ make
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver$ make deploy
```

8. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

9. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```


Listing 7-4: CY8C9520A_rpi3.c

```

#include <linux/i2c.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/gpio/driver.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>

#define DRV_NAME                "cy8c9520a"

/* cy8c9520a settings */
#define NGPIO                    20
#define DEVID_CY8C9520A        0x20
#define NPORTS                  3

/* Register offset */
#define REG_INPUT_PORT0         0x00
#define REG_OUTPUT_PORT0        0x08
#define REG_INTR_STAT_PORT0     0x10
#define REG_PORT_SELECT         0x18
#define REG_SELECT_PWM          0x1a
#define REG_INTR_MASK           0x19
#define REG_PIN_DIR             0x1c
#define REG_DRIVE_PULLUP        0x1d
#define REG_DRIVE_PULLDOWN      0x1e
#define REG_DEVID_STAT          0x2e

/* Definition of the global structure for the driver */
struct cy8c9520a {
    struct i2c_client *client;
    struct gpio_chip gpio_chip;
    struct gpio_desc *gpio;
    int irq;
    struct mutex lock;
    /* protect serialized access to the interrupt controller bus */
    struct mutex irq_lock;
    /* cached output registers */
    u8 outreg_cache[NPORTS];
    /* cached IRQ mask */
    u8 irq_mask_cache[NPORTS];
    /* IRQ mask to be applied */
    u8 irq_mask[NPORTS];
};

/* Per-port GPIO offset */
static const u8 cy8c9520a_port_offs[] = {
    0,
    8,
    16,
};

```

```

/* Return the port of the gpio */
static inline u8 cypress_get_port(unsigned int gpio)
{
    u8 i = 0;
    for (i = 0; i < sizeof(cy8c9520a_port_offs) - 1; i++) {
        if (! (gpio / cy8c9520a_port_offs[i + 1]))
            break;
    }
    return i;
}

/* Get the gpio offset inside its respective port */
static inline u8 cypress_get_offs(unsigned gpio, u8 port)
{
    return gpio - cy8c9520a_port_offs[port];
}

/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the REG_INPUT_PORT register
 */
static int cy8c9520a_gpio_get(struct gpio_chip *chip, unsigned int gpio)
{
    int ret;
    u8 port, in_reg;

    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    dev_info(chip->parent, "cy8c9520a_gpio_get function is called\n");

    /* Get the input port address (in_reg) for the GPIO */
    port = cypress_get_port(gpio);
    in_reg = REG_INPUT_PORT0 + port;

    dev_info(chip->parent, "the in_reg address is %u\n", in_reg);

    mutex_lock(&cygpio->lock);

    ret = i2c_smbus_read_byte_data(cygpio->client, in_reg);
    if (ret < 0) {
        dev_err(chip->parent, "can't read input port %u\n", in_reg);
    }

    dev_info(chip->parent, "cy8c9520a_gpio_get function with %d value is returned\n", ret);

    mutex_unlock(&cygpio->lock);

    /*
     * Check the status of the GPIO in its input port register
     * and return it. If expression is not 0 returns 1
     */
    return !(ret & BIT(cypress_get_offs(gpio, port)));
}

```

```

/*
 * struct gpio_chip set callback function.
 * It sets the output value of the GPIO line in
 * GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the REG_OUTPUT_PORT register
 */
static void cy8c9520a_gpio_set(struct gpio_chip *chip, unsigned int gpio, int val)
{
    int ret;
    u8 port, out_reg;
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    dev_info(chip->parent, "cy8c9520a_gpio_set_value func with %d value is called\n", val);

    /* Get the output port address (out_reg) for the GPIO */
    port = cypress_get_port(gpio);
    out_reg = REG_OUTPUT_PORT0 + port;

    mutex_lock(&cygpio->lock);

    /*
     * If val is 1, gpio output level is high.
     * If val is 0, gpio output level is low.
     * The output registers were previously cached in cy8c9520a_setup()
     */
    if (val) {
        cygpio->outreg_cache[port] |= BIT(cypress_get_offs(gpio, port));
    } else {
        cygpio->outreg_cache[port] &= ~BIT(cypress_get_offs(gpio, port));
    }

    ret = i2c_smbus_write_byte_data(cygpio->client, out_reg, cygpio->outreg_cache[port]);
    if (ret < 0) {
        dev_err(chip->parent, "can't write output port %u\n", port);
    }

    mutex_unlock(&cygpio->lock);
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO as an output writing to
 * the REG_PIN_DIR register of the selected port
 */
static int cy8c9520a_gpio_direction_output(struct gpio_chip *chip, unsigned int gpio, int val)
{
    int ret;
    u8 pins, port;

    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    /* Gets the port number of the gpio */
    port = cypress_get_port(gpio);

```

```

dev_info(chip->parent, "cy8c9520a_gpio_direction output is called\n");

mutex_lock(&cygpio->lock);

/* Select the port where we want to config the GPIO as an output */
ret = i2c_smbus_write_byte_data(cygpio->client, REG_PORT_SELECT, port);
if (ret < 0) {
    dev_err(chip->parent, "can't select port %u\n", port);
    goto err;
}

ret = i2c_smbus_read_byte_data(cygpio->client, REG_PIN_DIR);
if (ret < 0) {
    dev_err(chip->parent, "can't read pin direction\n");
    goto err;
}

/* Simply transform int to u8 */
pins = (u8)ret & 0xff;

/* Add the direction of the new pin. Set 1 for input and set 0 for output */
pins &= ~BIT(cypress_get_offs(gpio, port));

ret = i2c_smbus_write_byte_data(cygpio->client, REG_PIN_DIR, pins);
if (ret < 0) {
    dev_err(chip->parent, "can't write pin direction\n");
}

err:
mutex_unlock(&cygpio->lock);
cy8c9520a_gpio_set(chip, gpio, val);
return ret;
}

/*
 * struct gpio_chip direction_input callback function.
 * It configures the GPIO as an input writing to
 * the REG_PIN_DIR register of the selected port
 */
static int cy8c9520a_gpio_direction_input(struct gpio_chip *chip, unsigned int gpio)
{
    int ret;
    u8 pins, port;

    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    /* Gets the port number of the gpio */
    port = cypress_get_port(gpio);

    dev_info(chip->parent, "cy8c9520a_gpio_direction input is called\n");

    mutex_lock(&cygpio->lock);

```

```

/* Select the port where we want to config the GPIO as input */
ret = i2c_smbus_write_byte_data(cygpio->client, REG_PORT_SELECT, port);
if (ret < 0) {
    dev_err(chip->parent, "can't select port %u\n", port);
    goto err;
}

ret = i2c_smbus_read_byte_data(cygpio->client, REG_PIN_DIR);
if (ret < 0) {
    dev_err(chip->parent, "can't read pin direction\n");
    goto err;
}

/* Simply transform int to u8 */
pins = (u8)ret & 0xff;

/*
 * Add the direction of the new pin.
 * Set 1 for input (out == 0) and set 0 for output (out == 1)
 */
pins |= BIT(cypress_get_offs(gpio, port));

ret = i2c_smbus_write_byte_data(cygpio->client, REG_PIN_DIR, pins);
if (ret < 0) {
    dev_err(chip->parent, "can't write pin direction\n");
    goto err;
}

err:
    mutex_unlock(&cygpio->lock);
    return ret;
}

/* Function to lock access to slow bus (i2c) chips */
static void cy8c9520a_irq_bus_lock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    dev_info(chip->parent, "cy8c9520a_irq_bus_lock is called\n");
    mutex_lock(&cygpio->irq_lock);
}

/*
 * Function to sync and unlock slow bus (i2c) chips.
 * REG_INTR_MASK register is accessed via I2C.
 * Write 0 to the interrupt mask register line to
 * activate the interrupt on the GPIO
 */
static void cy8c9520a_irq_bus_sync_unlock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    int ret, i;
    unsigned int gpio;

```

```

u8 port;
dev_info(chip->parent, "cy8c9520a_irq_bus_sync_unlock is called\n");

gpio = d->hwirq;
port = cypress_get_port(gpio);

/* irq_mask_cache stores the last value of irq_mask for each port */
for (i = 0; i < NPORTS; i++) {
    /*
     * Check if some of the bits have changed from the last cached value.
     * irq_mask registers were initialized in cy8c9520a_irq_setup()
     */
    if (cygpio->irq_mask_cache[i] ^ cygpio->irq_mask[i]) {
        dev_info(chip->parent, "gpio %u is unmasked\n", gpio);
        cygpio->irq_mask_cache[i] = cygpio->irq_mask[i];
        ret = i2c_smbus_write_byte_data(cygpio->client,
                                         REG_PORT_SELECT, i);

        if (ret < 0) {
            dev_err(chip->parent, "can't select port %u\n", port);
            goto err;
        }

        /* Enable the interrupt for the GPIO unmasked */
        ret = i2c_smbus_write_byte_data(cygpio->client, REG_INTR_MASK,
                                         cygpio->irq_mask[i]);

        if (ret < 0) {
            dev_err(chip->parent,
                    "can't write int mask on port %u\n", port);
            goto err;
        }

        ret = i2c_smbus_read_byte_data(cygpio->client, REG_INTR_MASK);
        dev_info(chip->parent, "the REG_INTR_MASK value is %d\n", ret);
    }
}

err:
    mutex_unlock(&cygpio->irq_lock);
}

/*
 * Mask (disable) the GPIO interrupt.
 * In the initial setup all the INT lines are masked
 */
static void cy8c9520a_irq_mask(struct irq_data *d)
{
    u8 port;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    unsigned gpio = d->hwirq;
    port = cypress_get_port(gpio);
    dev_info(chip->parent, "cy8c9520a_irq_mask is called\n");

    cygpio->irq_mask[port] |= BIT(cypress_get_offs(gpio, port));
}

```

```

}

/*
 * Unmask (enable) the GPIO interrupt.
 * In the initial setup all the INT lines are masked
 */
static void cy8c9520a_irq_unmask(struct irq_data *d)
{
    u8 port;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    unsigned gpio = d->hwirq;
    port = cypress_get_port(gpio);
    dev_info(chip->parent, "cy8c9520a_irq_unmask is called\n");

    cygpio->irq_mask[port] &= ~BIT(cypress_get_offs(gpio, port));
}

/* Set the flow type of the IRQ */
static int cy8c9520a_irq_set_type(struct irq_data *d, unsigned int type)
{
    int ret = 0;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    dev_info(chip->parent, "cy8c9520a_irq_set_type is called\n");

    if ((type != IRQ_TYPE_EDGE_BOTH) && (type != IRQ_TYPE_EDGE_FALLING)) {
        dev_err(&cygpio->client->dev, "irq %d: unsupported type %d\n", d->irq, type);
        ret = -EINVAL;
        goto err;
    }

err:
    return ret;
}

/* Initialization of the irq_chip structure with callback functions */
static struct irq_chip cy8c9520a_irq_chip = {
    .name           = "cy8c9520a-irq",
    .irq_mask       = cy8c9520a_irq_mask,
    .irq_unmask     = cy8c9520a_irq_unmask,
    .irq_bus_lock   = cy8c9520a_irq_bus_lock,
    .irq_bus_sync_unlock = cy8c9520a_irq_bus_sync_unlock,
    .irq_set_type   = cy8c9520a_irq_set_type,
};

/*
 * Interrupt handler for the cy8c9520a. It is called when
 * there is a rising or falling edge in the unmasked GPIO
 */
static irqreturn_t cy8c9520a_irq_handler(int irq, void *devid)
{
    struct cy8c9520a *cygpio = devid;

```

```

u8 stat[NPORTS], pending;
unsigned port, gpio, gpio_irq;
int ret;

pr_info ("the interrupt ISR has been entered\n");

/*
 * Store in stat and clear (to enable INTs)
 * the three interrupt status registers by reading them
 */
ret = i2c_smbus_read_i2c_block_data(cygpio->client, REG_INTR_STAT_PORT0, NPORTS, stat);
if (ret < 0) {
    memset(stat, 0, sizeof(stat));
}

ret = IRQ_NONE;

for (port = 0; port < NPORTS; port++) {
    mutex_lock(&cygpio->irq_lock);

    /*
     * In every port, check the GPIOs that have their INT unmasked
     * and whose bits have been enabled in their REG_INTR_STAT_PORT
     * register due to an interrupt in the GPIO, then store the new
     * value in the pending register
     */
    pending = stat[port] & (~cygpio->irq_mask[port]);
    mutex_unlock(&cygpio->irq_lock);

    /* Launch the ISRs of all the gpios that requested an interrupt */
    while (pending) {
        ret = IRQ_HANDLED;
        /* get the first gpio that has got an int */
        gpio = __ffs(pending);

        /* clears the gpio in the pending register */
        pending &= ~BIT(gpio);

        /* gets the int number associated to this gpio */
        gpio_irq = cy8c9520a_port_offs[port] + gpio;

        /* launch the ISR of the GPIO child driver */
        handle_nested_irq(irq_find_mapping(cygpio->gpio_chip.irq.domain,
                                           gpio_irq));
    }
}

return ret;
}

/* Initial setup for the cy8c9520a */
static int cy8c9520a_setup(struct cy8c9520a *cygpio)
{

```



```

int ret, i;
struct i2c_client *client = cygprio->client;

/* Disable PWM, set all GPIOs as input. */
for (i = 0; i < NPORTS; i++) {
    ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
    if (ret < 0) {
        dev_err(&client->dev, "can't select port %u\n", i);
        goto end;
    }

    ret = i2c_smbus_write_byte_data(client, REG_SELECT_PWM, 0x00);
    if (ret < 0) {
        dev_err(&client->dev, "can't write to SELECT_PWM\n");
        goto end;
    }

    ret = i2c_smbus_write_byte_data(client, REG_PIN_DIR, 0xff);
    if (ret < 0) {
        dev_err(&client->dev, "can't write to PIN_DIR\n");
        goto end;
    }
}

/* Cache the output registers (Output Port 0, Output Port 1, Output Port 2) */
ret = i2c_smbus_read_i2c_block_data(client, REG_OUTPUT_PORT0,
                                     sizeof(cygprio->outreg_cache),
                                     cygprio->outreg_cache);

if (ret < 0) {
    dev_err(&client->dev, "can't cache output registers\n");
    goto end;
}

dev_info(&client->dev, "the cy8c9520a_setup is done\n");

end:
    return ret;
}

/* Interrupt setup for the cy8c9520a */
static int cy8c9520a_irq_setup(struct cy8c9520a *cygprio)
{
    struct i2c_client *client = cygprio->client;
    struct gpio_chip *chip = &cygprio->gpio_chip;
    u8 dummy[NPORTS];
    int ret, i;

    mutex_init(&cygprio->irq_lock);
    dev_info(&client->dev, "the cy8c9520a_irq_setup function is entered\n");

    /*
     * Clear interrupt state registers by reading the three registers:
     * Interrupt Status Port0, Interrupt Status Port1, Interrupt Status Port2,
     * and store the values in a dummy array
    */

```

```

    */
    ret = i2c_smbus_read_i2c_block_data(client, REG_INTR_STAT_PORT0, NPORTS, dummy);
    if (ret < 0) {
        dev_err(&client->dev, "couldn't clear int status\n");
        goto err;
    }

    dev_info(&client->dev, "the interrupt state registers are cleared\n");

    /*
     * Initialize Interrupt Mask Port Register (19h) for each port.
     * Disable the activation of the INT lines. Each 1 in this
     * register, masks (disables) the INT from the corresponding GPIO
     */
    memset(cygpio->irq_mask_cache, 0xff, sizeof(cygpio->irq_mask_cache));
    memset(cygpio->irq_mask, 0xff, sizeof(cygpio->irq_mask));

    /* Disable interrupts in all the gpio lines */
    for (i = 0; i < NPORTS; i++) {
        ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
        if (ret < 0) {
            dev_err(&client->dev, "can't select port %u\n", i);
            goto err;
        }

        ret = i2c_smbus_write_byte_data(client, REG_INTR_MASK, cygpio->irq_mask[i]);
        if (ret < 0) {
            dev_err(&client->dev, "can't write int mask on port %u\n", i);
            goto err;
        }
    }

    dev_info(&client->dev, "the interrupt mask port registers are set\n");

    /* Add a nested irqchip to the gpiochip */
    ret = gpiochip_irqchip_add_nested(chip,
                                     &cy8c9520a_irq_chip,
                                     0,
                                     handle_simple_irq,
                                     IRQ_TYPE_NONE);

    if (ret) {
        dev_err(&client->dev, "could not connect irqchip to gpiochip\n");
        return ret;
    }

    /*
     * Request interrupt on a GPIO pin of the external processor.
     * This processor pin is connected to the INT pin of the cy8c9520a
     */
    ret = devm_request_threaded_irq(&client->dev, client->irq, NULL,
                                   cy8c9520a_irq_handler,
                                   IRQF_ONESHOT | IRQF_TRIGGER_HIGH,
                                   dev_name(&client->dev), cygpio);

    if (ret) {

```

```

        dev_err(&client->dev, "failed to request irq %d\n", cygpio->irq);
        return ret;
    }

    /*
     * Set up a nested irq handler for a gpio_chip from a parent IRQ.
     * You can now request interrupts from GPIO child drivers nested
     * to the cy8c9520a driver
     */
    gpiochip_set_nested_irqchip(chip, &cy8c9520a_irq_chip, client->irq);

    dev_info(&client->dev, "the interrupt setup is done\n");

    return 0;
err:
    mutex_destroy(&cygpio->irq_lock);
    return ret;
}

/*
 * Initialize the cy8c9520a gpio controller (struct gpio_chip)
 * and register it to the kernel
 */
static int cy8c9520a_gpio_init(struct cy8c9520a *cygpio)
{
    struct gpio_chip *gpiochip = &cygpio->gpio_chip;
    int err;

    gpiochip->label = cygpio->client->name;
    gpiochip->base = -1;
    gpiochip->ngpio = NGPIO;
    gpiochip->parent = &cygpio->client->dev;
    gpiochip->of_node = gpiochip->parent->of_node;
    gpiochip->can_sleep = true;
    gpiochip->direction_input = cy8c9520a_gpio_direction_input;
    gpiochip->direction_output = cy8c9520a_gpio_direction_output;
    gpiochip->get = cy8c9520a_gpio_get;
    gpiochip->set = cy8c9520a_gpio_set;
    gpiochip->owner = THIS_MODULE;

    /* Register a gpio_chip */
    err = devm_gpiochip_add_data(gpiochip->parent, gpiochip, cygpio);
    if (err)
        return err;
    return 0;
}

static int cy8c9520a_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    struct cy8c9520a *cygpio;
    int ret;
    unsigned int dev_id;

    dev_info(&client->dev, "cy8c9520a_probe() function is called\n");

```

```

if (!i2c_check_functionality(client->adapter,
                             I2C_FUNC_SMBUS_I2C_BLOCK | I2C_FUNC_SMBUS_BYTE_DATA)) {
    dev_err(&client->dev, "SMBUS Byte/Block unsupported\n");
    return -EIO;
}

/* Allocate global private structure for a new device */
cygpio = devm_kzalloc(&client->dev, sizeof(*cygpio), GFP_KERNEL);
if (!cygpio) {
    dev_err(&client->dev, "failed to alloc memory\n");
    return -ENOMEM;
}

cygpio->client = client;

mutex_init(&cygpio->lock);

/* Whoami */
dev_id = i2c_smbus_read_byte_data(client, REG_DEVID_STAT);
if (dev_id < 0) {
    dev_err(&client->dev, "can't read device ID\n");
    ret = dev_id;
    goto err;
}
dev_info(&client->dev, "dev_id=0x%x\n", dev_id & 0xff);

/* Initial setup for the cy8c9520a */
ret = cy8c9520a_setup(cygpio);
if (ret < 0) {
    goto err;
}

/* Initialize the cy8c9520a gpio controller */
ret = cy8c9520a_gpio_init(cygpio);
if (ret) {
    goto err;
}

/* Interrupt setup for the cy8c9520a */
ret = cy8c9520a_irq_setup(cygpio);
if (ret) {
    goto err;
}

/* Link the I2C device with the cygpio device */
i2c_set_clientdata(client, cygpio);

return 0;
err:
mutex_destroy(&cygpio->lock);

return ret;
}

```

```

static int cy8c9520a_remove(struct i2c_client *client)
{
    dev_info(&client->dev, "cy8c9520a_remove() function is called\n");
    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "cy8c9520a"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static const struct i2c_device_id cy8c9520a_id[] = {
    {DRV_NAME, 0},
    {}},
};
MODULE_DEVICE_TABLE(i2c, cy8c9520a_id);

static struct i2c_driver cy8c9520a_driver = {
    .driver = {
        .name = DRV_NAME,
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    },
    .probe = cy8c9520a_probe,
    .remove = cy8c9520a_remove,
    .id_table = cy8c9520a_id,
};
module_i2c_driver(cy8c9520a_driver);

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a driver that controls the \
    cy8c9520a I2C GPIO expander");

```

LAB 7.4 GPIO child driver description

In this lab, you will develop a GPIO child driver (`int_rpi3_gpio`) which will request a GPIO interrupt from the CY8C9520A gpio controller. You will use the LAB 7.1 module as a starting point for the development of the driver. Whenever there is a level change in the first input line of the CY8C9520A P0 port, an interrupt output (INT) will be asserted by the CY8C9520A GPIO controller, and its interrupt handler `cy8c9520a_irq_handler()` will be called. The interrupt handler `cy8c9520a_irq_handler()` will call `handle_nested_irq()`, which in turn calls the interrupt handler `P0_line0_isr()` of our GPIO child driver.

The GPIO child driver will request the GPIO INT by using the `devm_request_threaded_irq()` function. Before calling this function, the driver will return the Linux IRQ number from the Device Tree by using the `platform_get_irq()` function.

Open the `bcm2710-rpi-3-b.dts` DT file, and add the `int_gpio` node below the `int_key` one. Check the differences between both Device Tree nodes. In our new driver, the `interrupt-parent` is the `cy8c9520a` node of our `CY8C9520A` gpio controller driver, and the GPIO interrupt line included in the `interrupts` property has the number 0, which matches with the first input line of the `CY8C9520A` P0 controller.

```
int_key {
    compatible = "arrow,intkey";
    pinctrl-names = "default";
    pinctrl-0 = <&key_pin>;
    gpios = <&gpio 23 0>;
    interrupts = <23 1>;
    interrupt-parent = <&gpio>;
};

int_gpio {
    compatible = "arrow,int_gpio_expand";
    pinctrl-names = "default";
    interrupt-parent = <&cy8c9520a>;
    interrupts = <0 IRQ_TYPE_EDGE_BOTH>;
};
```

Create a new `linux_5.4_gpio_int_driver` folder inside the `linux_5.4_CY8C9520A_driver` folder:

```
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver$ mkdir linux_5.4_gpio_int_driver
```

Create a new `int_rpi3_gpio.c` file and a Makefile file in the `linux_5.4_gpio_int_driver` folder, and add `int_rpi3_gpio.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver/linux_5.4_gpio_int_driver$ make
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver/linux_5.4_gpio_int_driver$ make deploy
```

Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 7-5: int_rpi3_gpio.c

```

#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/interrupt.h>
#include <linux/gpio/consumer.h>
#include <linux/miscdevice.h>
#include <linux/of_device.h>

static char *INT_NAME = "P0_line0_INT";

/* Interrupt handler */
static irqreturn_t P0_line0_isr(int irq, void *data)
{
    struct device *dev = data;
    dev_info(dev, "interrupt received. key: %s\n", INT_NAME);
    return IRQ_HANDLED;
}

static struct miscdevice helloworld_miscdevice = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "mydev",
};

static int my_probe(struct platform_device *pdev)
{
    int ret_val, irq;
    struct device *dev = &pdev->dev;

    dev_info(dev, "my_probe() function is called.\n");

    /* Get the Linux IRQ number */
    irq = platform_get_irq(pdev, 0);
    if (irq < 0){
        dev_err(dev, "irq is not available\n");
        return -EINVAL;
    }
    dev_info(dev, "IRQ_using_platform_get_irq: %d\n", irq);

    /* Allocate the interrupt line */
    ret_val = devm_request_threaded_irq(dev, irq, NULL, P0_line0_isr,
                                         IRQF_ONESHOT | IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
                                         INT_NAME, dev);

    if (ret_val) {
        dev_err(dev, "Failed to request interrupt %d, error %d\n", irq, ret_val);
        return ret_val;
    }

    ret_val = misc_register(&helloworld_miscdevice);
    if (ret_val != 0)
    {
        dev_err(dev, "could not register the misc device mydev\n");
        return ret_val;
    }
}

```

```

    }

    dev_info(dev, "mydev: got minor %i\n", helloworld_miscdevice.minor);
    dev_info(dev, "my_probe() function is exited.\n");

    return 0;
}

static int my_remove(struct platform_device *pdev)
{
    dev_info(&pdev->dev, "my_remove() function is called.\n");
    misc_deregister(&helloworld_miscdevice);
    dev_info(&pdev->dev, "my_remove() function is exited.\n");
    return 0;
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,int_gpio_expand",
      {} },
};

MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "int_gpio_expand",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};

module_platform_driver(my_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a GPIO INT platform driver");

```

LAB 7.4 GPIO based IRQ application

In the previous section, you have seen how to request and handle a GPIO INT using a GPIO child driver. In the following Listing 7-6, you will see how to request and handle an interrupt from user space for the first line of the CY8C9520A P0 port. You will use the GPIOLib user space API, which handles the GPIO INT through ioctl calls on the /dev/gpiochip3 char device file.

Create a new app folder inside the linux_5.4_CY8C9520A_driver folder:

```
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver$ mkdir app
```

Create a new gpio_int.c file in the app folder, and send the application to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_driver/app$ scp gpio_int.c root@10.0.0.10:/home/pi
```


Compile the application on the Raspberry Pi:

```
root@raspberrypi:/home/pi# gcc -o gpio_int gpio_int.c
```

Listing 7-6: gpio_int.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <poll.h>
#include <string.h>
#include <linux/gpio.h>
#include <sys/ioctl.h>

#define DEV_GPIO  "/dev/gpiochip3"

#define POLL_TIMEOUT -1 /* No timeout */

int main(int argc, char *argv[])
{
    int fd, fd_in;
    int ret;
    int flags;

    struct gpioevent_request req;
    struct gpioevent_data evdata;
    struct pollfd fdset;

    /* Open gpio */
    fd = open(DEV_GPIO, O_RDWR);
    if (fd < 0) {
        printf("ERROR: open %s ret=%d\n", DEV_GPIO, fd);
        return -1;
    }

    /* Request GPIO P0 first line interrupt */
    req.lineoffset = 0;
    req.handleflags = GPIOHANDLE_REQUEST_INPUT;
    req.eventflags = GPIOEVENT_REQUEST_BOTH_EDGES;
    strncpy(req.consumer_label, "gpio_irq", sizeof(req.consumer_label) - 1);

    /* Request line event handle */
    ret = ioctl(fd, GPIO_GET_LINEEVENT_IOCTL, &req);
    if (ret) {
        printf("ERROR: ioctl get line event ret=%d\n", ret);
        return -1;
    }

    /* Set event fd nonblock read */
    fd_in = req.fd;
    flags = fcntl(fd_in, F_GETFL);
```

```
flags |= O_NONBLOCK;
ret = fcntl(fd_in, F_SETFL, flags);
if (ret) {
    printf("ERROR: fcntl set nonblock read\n");
}

for (;;) {
    fdset.fd      = fd_in;
    fdset.events  = POLLIN;
    fdset.revents = 0;

    /* poll gpio line event */
    ret = poll(&fdset, 1, POLL_TIMEOUT);
    if (ret <= 0)
        continue;

    if (fdset.revents & POLLIN) {
        printf("irq received.\n");
        /* read event data */
        ret = read(fd_in, &evdata, sizeof(evdata));
        if (ret == sizeof(evdata))
            printf("id: %d, timestamp: %lld\n", evdata.id, evdata.timestamp);
    }
}

/* close gpio */
close(fd);

return 0;
}
```

LAB 7.4 driver demonstration

Connect your Raspberry Pi to the Internet and download libgpiod library and tools:

```
root@raspberrypi:/home# sudo apt-get install gpiod libgpiod-dev libgpiod-doc
```

Scan I2C bus for devices:

```
root@raspberrypi:/home/pi# i2cdetect -y 1
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20: 20  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```

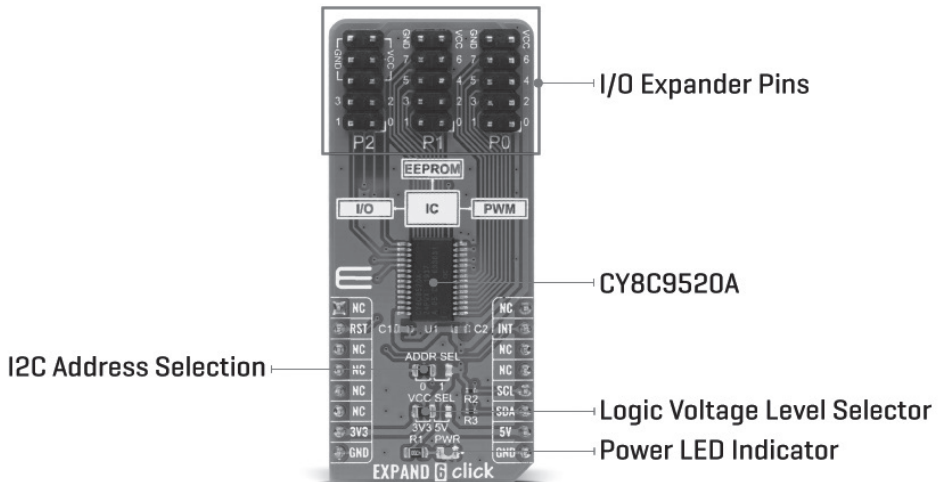
Load the CY8C9520A module:

```
root@raspberrypi:/home/pi# insmod CY8C9520A_rpi3.ko
cy8c9520a 1-0020: cy8c9520a_probe() function is called
cy8c9520a 1-0020: dev_id=0x20
cy8c9520a 1-0020: the cy8c9520a_setup is done
cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
cy8c9520a 1-0020: the interrupt state registers are cleared
cy8c9520a 1-0020: the interrupt mask port registers are set
cy8c9520a 1-0020: the interrupt setup is done
```

Print information of all the lines of the gpiochip3:

```
root@raspberrypi:/home/pi# gpioinfo gpiochip3
gpiochip3 - 20 lines:
    line 0:      unnamed      unused  input  active-high
    line 1:      unnamed      unused  input  active-high
    line 2:      unnamed      unused  input  active-high
    line 3:      unnamed      unused  input  active-high
    line 4:      unnamed      unused  input  active-high
    line 5:      unnamed      unused  input  active-high
    line 6:      unnamed      unused  input  active-high
    line 7:      unnamed      unused  input  active-high
    line 8:      unnamed      unused  input  active-high
    line 9:      unnamed      unused  input  active-high
    line 10:     unnamed      unused  input  active-high
    line 11:     unnamed      unused  input  active-high
    line 12:     unnamed      unused  input  active-high
    line 13:     unnamed      unused  input  active-high
    line 14:     unnamed      unused  input  active-high
    line 15:     unnamed      unused  input  active-high
    line 16:     unnamed      unused  input  active-high
    line 17:     unnamed      unused  input  active-high
    line 18:     unnamed      unused  input  active-high
    line 19:     unnamed      unused  input  active-high
```

Connect pin 0 to pin 1 on the P0 port of the I/O Expander board. The gpio lines of the gpiochip3 are configured with internal pull-up to Vcc. See below the I/O Expander Pins of the CY8C9520A board:



Set pin 1 of P0 to high:

```
root@raspberrypi:/home/pi# gpioset gpiochip3 1=1
cy8c9520a 1-0020: cy8c9520a_gpio_direction output is called
cy8c9520a 1-0020: cy8c9520a_gpio_set_value func with 1 value is called
```

Read pin 0 of P0:

```
root@raspberrypi:/home/pi# gpioget gpiochip3 0
cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
cy8c9520a 1-0020: the in_reg address is 0
cy8c9520a 1-0020: cy8c9520a_gpio_get function with 255 value is returned
1
```

Set pin 1 of P0 to low:

```
root@raspberrypi:/home/pi# gpioset gpiochip3 1=0
cy8c9520a 1-0020: cy8c9520a_gpio_direction output is called
cy8c9520a 1-0020: cy8c9520a_gpio_set_value func with 0 value is called
```

Read pin 0 of P0:

```
root@raspberrypi:/home/pi# gpioget gpiochip3 0
cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
```

```
cy8c9520a 1-0020: the in_reg address is 0
cy8c9520a 1-0020: cy8c9520a_gpio_get function with 252 value is returned
0
```

Disconnect pin 0 and pin 1 on the P0 port of the I/O Expander pins. Handle GPIO INT in line 0 of P0 using the gpio interrupt driver.

Load the gpio interrupt module:

```
root@raspberrypi:/home/pi# insmod int_rpi3_gpio.ko
int_gpio_expand soc:int_gpio: my_probe() function is called.
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
cy8c9520a 1-0020: gpio 0 is unmasked
the interrupt ISR has been entered
cy8c9520a 1-0020: the REG_INTR_MASK value is 254
int_gpio_expand soc:int_gpio: IRQ_using_platform_get_irq: 167
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
int_gpio_expand soc:int_gpio: mydev: got minor 60
int_gpio_expand soc:int_gpio: my_probe() function is exited.
```

See the gpio interrupt with Linux IRQ number 167 and hardware irq 0 for the cy8c9520a-irq controller:

```
root@raspberrypi:/home/pi# cat /proc/interrupts
CPU0          CPU1          CPU2          CPU3
 17:           878             0             0             0  ARMCTRL-level   1 Edge       3f
00b880.mailbox
 18:           720             0             0             0  ARMCTRL-level   2 Edge       VC
[...]
167:           0             0             0             0  cy8c9520a-irq   0 Edge       P0
_line0_INT
FIQ:                usb_fiq
IPI0:           0             0             0             0  CPU wakeup interrupts
IPI1:           0             0             0             0  Timer broadcast interrupts
IPI2:          3927          8767          6914          7539  Rescheduling interrupts
IPI3:           416          1136          1334          1307  Function call interrupts
IPI4:           0             0             0             0  CPU stop interrupts
IPI5:          1568          2207          450           933  IRQ work interrupts
IPI6:           0             0             0             0  completion interrupts
Err:           0
```

Connect pin 0 of P0 to GND, then disconnect it from GND. Two interrupts are fired:

```
root@raspberrypi:/home/pi#
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
```

Remove the gpio int module:

```
root@raspberrypi:/home/pi# rmmod int_rpi3_gpio.ko
```

```

int_gpio_expand int_gpio: my_remove() function is called.
int_gpio_expand int_gpio: my_remove() function is exited.
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_mask is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called

```

Remove the CY8C9520A module:

```

root@raspberrypi:/home/pi# rmmod CY8C9520A_rpi3.ko
cy8c9520a 1-0020: cy8c9520a_remove() function is called

```

Reboot the system. Handle GPIO INT in line 0 of P0 using a GPIO based interrupt application.

```

root@raspberrypi:/home/pi# reboot

```

Load the CY8C9520A module:

```

root@raspberrypi:/home/pi# insmod CY8C9520A_rpi3.ko
CY8C9520A_rpi4: loading out-of-tree module taints kernel.
cy8c9520a 1-0020: cy8c9520a_probe() function is called
cy8c9520a 1-0020: dev_id=0x20
cy8c9520a 1-0020: the cy8c9520a_setup is done
cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
cy8c9520a 1-0020: the interrupt state registers are cleared
cy8c9520a 1-0020: the interrupt mask port registers are set
cy8c9520a 1-0020: the interrupt setup is done

```

Launch the gpiomon application:

```

root@raspberrypi:/home/pi# gpiomon --falling-edge gpiochip3 0
cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
cy8c9520a 1-0020: gpio 0 is unmasked
cy8c9520a 1-0020: the REG_INTR_MASK value is 254
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called

```

Connect pin 0 of P0 to GND. An interrupt is fired:

```

the interrupt ISR has been entered
event: FALLING EDGE offset: 0 timestamp: [1606046305.553936360]

```

Disconnect pin 0 of P0 from GND. An interrupt is fired:

```

the interrupt ISR has been entered
event: FALLING EDGE offset: 0 timestamp: [1606046308.068990655]

```

Exit the application with ^C:

```

root@raspberrypi:/home/pi#

```

Launch the `gpio_int` application. Connect pin 0 of P0 to GND, then remove it from GND. Two interrupts are fired:

```
root@raspberrypi:/home/pi# ./gpio_int
cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
the interrupt ISR has been entered
cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
cy8c9520a 1-0020: the in_reg address is 0
cy8c9520a 1-0020: cy8c9520a_gpio_get function with 254 value is returned
irq received.
id: 2, timestamp: 1606046401377764044
the interrupt ISR has been entered
cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
cy8c9520a 1-0020: the in_reg address is 0
cy8c9520a 1-0020: cy8c9520a_gpio_get function with 255 value is returned
irq received.
id: 1, timestamp: 1606046404416847616
```

Exit the application with `^C`:

```
root@raspberrypi:/home/pi#
```

Remove the `CY8C9520A` module:

```
root@raspberrypi:/home/pi# rmmod CY8C9520A_rpi3.ko
cy8c9520a 1-0020: cy8c9520a_remove() function is called
```

LAB 7.5: "GPIO-PWM-PINCTRL expander device" module

The `CY8C9520A_pwm_pinctrl` driver that you will develop in this LAB 7.5 is an extension of the previous `CY8C9520A_rpi3` driver, to which you will add new "pin controller" and "PWM controller" capabilities.

LAB 7.5 GPIO irqchip description

In the LAB 7.4, you developed a driver for an off-chip GPIO expander with interrupt capabilities that uses the `gpiochip_irqchip_add_nested()` function. This way of adding a nested cascaded irqchip to a gpiochip is still available in kernel 5.4, but the preferred way to set up the helpers is to fill in the `gpio_irq_chip` structure that is inside `struct gpio_chip` before adding `gpio_chip` to the kernel by using `devm_gpiochip_add_data()`. If you do this, the additional `irq_chip` will be set up by `gpiolib` at the same time as setting up the rest of the GPIO functionality. In this LAB 7.5, you will develop a driver for the same GPIO expander of the LAB 7.4, but this time you will use this new method. You will follow the next steps:

1. You will include an `irq_chip` structure in the `cy8c9520a` private structure:

```
/* definition of the global structure for the driver */
struct cy8c9520a {
    struct i2c_client    *client;
    struct gpio_chip     gpio_chip;
    struct irq_chip      irq_chip;
    struct pwm_chip      pwm_chip;
    struct gpio_desc     *gpio;
    int                  irq;
    struct mutex          lock;
    /* protect serialized access to the interrupt controller bus */
    struct mutex          irq_lock;
    /* cached output registers */
    u8                    outreg_cache[NPORTS];
    /* cached IRQ mask */
    u8                    irq_mask_cache[NPORTS];
    /* IRQ mask to be applied */
    u8                    irq_mask[NPORTS];
    int                   pwm_number[NPWM];

    struct pinctrl_dev    *pctldev;
    struct pinctrl_desc   pinctrl_desc;
};
```

2. You will add the next lines of code in bold to fill the `gpio_irq_chip` structure that is inside `struct gpio_chip`:

```
/* Interrupt setup for the cy8c9520a */
static int cy8c9520a_irq_setup(struct cy8c9520a *cygpio)
{
    struct gpio_irq_chip *girq;
    struct i2c_client *client = cygpio->client;

    [...]

    /*
     * Request interrupt on a GPIO pin of the processor
     * this processor GPIO is connected to the INT pin of cy8c9520a
     */
    ret = devm_request_threaded_irq(&client->dev, client->irq, NULL,
                                   cy8c9520a_irq_handler,
                                   IRQF_ONESHOT | IRQF_TRIGGER_HIGH,
                                   dev_name(&client->dev), cygpio);

    if (ret) {
        dev_err(&client->dev, "failed to request irq %d\n", cygpio->irq);
        return ret;
    }

    /*
     * set up a nested irq handler for a gpio_chip from a parent IRQ
     * you can now request interrupts from GPIO child drivers nested
     * to the cy8c9520a driver
     */
```



```

        girq = &cygpio->gpio_chip.irq;
        girq->chip = &cy8c9520a_irq_chip;
        girq->parent_handler = NULL;
        girq->num_parents = 0;
        girq->parents = NULL;
        girq->default_type = IRQ_TYPE_NONE;
        girq->handler = handle_simple_irq;
        girq->threaded = true;

        [...]
    }

```

3. Finally, you will register the `gpio_chip` with the kernel by using the `devm_gpiochip_add_data()` function:

```
devm_gpiochip_add_data(&client->dev, &cygpio->gpio_chip, cygpio);
```

LAB 7.5 pin controller driver description

As described in Chapter 5 of this book, a pin controller is a peripheral of the processor that can configure pin hardware settings. It may be able to multiplex, bias, set load capacitance and set drive modes (pull up or down, open drain high/low, strong drive fast/slow, or high-impedance input) for individual pins or groups of pins. The pin controller section of this driver will configure several drive modes for the CY8C9520A port data pins (pull up, pull down and strong drive). On the software side, the Linux Pinctrl framework configures and controls the pins. There are two ways to use it:

- The Pinctrl framework will apply the pin configuration given by the Device Tree by calling specific vendor callback functions. This is the way that we will use in our driver.
- A pin needs to be controlled by software. The GPIOLib framework will be used to control this pin on top of the Pinctrl framework. For GPIOs that use pins known to the Pinctrl subsystem, that subsystem should be informed of their use. A gpiolib driver's `.request()` operation may call `pinctrl_request_gpio()`, and a gpiolib driver's `.free()` operation may call `pinctrl_free_gpio()`. The Pinctrl subsystem allows a `pinctrl_request_gpio()` to succeed concurrently with a pin or pingroup being "owned" by a device for pin multiplexing. The `gpio` and `pin` controllers are associated with each other through the `pinctrl_add_gpio_range()` function, which adds a range of GPIOs to be handled by a certain pin controller.

The first step during the development of our driver's `pinctrl` code is to tell the `pinctrl` framework which pins the CY8C9520A device provides; that is a simple matter of enumerating their names and associating each with an integer pin number. You will create a `pinctrl_pin_desc` structure with the unique pin numbers from the global pin number space and the name for these pins. You have to use these names when you configure your Device Tree pin configuration nodes.

```
static const struct pinctrl_pin_desc cy8c9520a_pins[] = {
    PINCTRL_PIN(0, "gpio0"),
    PINCTRL_PIN(1, "gpio1"),
    PINCTRL_PIN(2, "gpio2"),
    PINCTRL_PIN(3, "gpio3"),
    PINCTRL_PIN(4, "gpio4"),
    PINCTRL_PIN(5, "gpio5"),
    PINCTRL_PIN(6, "gpio6"),
    PINCTRL_PIN(7, "gpio7"),
    PINCTRL_PIN(8, "gpio8"),
    PINCTRL_PIN(9, "gpio9"),
    PINCTRL_PIN(10, "gpio10"),
    PINCTRL_PIN(11, "gpio11"),
    PINCTRL_PIN(12, "gpio12"),
    PINCTRL_PIN(13, "gpio13"),
    PINCTRL_PIN(14, "gpio14"),
    PINCTRL_PIN(15, "gpio15"),
    PINCTRL_PIN(16, "gpio16"),
    PINCTRL_PIN(17, "gpio17"),
    PINCTRL_PIN(18, "gpio18"),
    PINCTRL_PIN(19, "gpio19"),
};
```

A pin controller is registered to the Pinctrl subsystem by filling in a `pinctrl_desc` structure and calling the `devm_pinctrl_register()` function, which takes the `pinctrl_desc` structure as an argument. See below the setup of the `pintrl_desc` structure, done in our driver's `probe()` function.

```
cygpio->pinctrl_desc.name = "cy8c9520a-pinctrl";
cygpio->pinctrl_desc.pctlops = &cygpio_pinctrl_ops;
cygpio->pinctrl_desc.confops = &cygpio_pinconf_ops;
cygpio->pinctrl_desc.npins = cygpio->gpio_chip.ngpio;

cygpio->pinctrl_desc.pins = cy8c9520a_pins;
cygpio->pinctrl_desc.owner = THIS_MODULE;

cygpio->pctldev = devm_pinctrl_register(&client->dev, &cygpio->pinctrl_desc, cygpio);
```

The `pctlops` variable points to the custom `cygpio_pinctrl_ops` structure, which contains pointers to several callback functions. The `pinconf_generic_dt_node_to_map_pin` function parses our Device Tree "pin configuration nodes" and creates mapping table entries for them. You will not implement the rest of the callback functions inside the `pinctrl_ops` structure.

```
static const struct pinctrl_ops cygpio_pinctrl_ops = {
    .get_groups_count = cygpio_pinctrl_get_groups_count,
    .get_group_name = cygpio_pinctrl_get_group_name,
    .get_group_pins = cygpio_pinctrl_get_group_pins,
#ifdef CONFIG_OF
    .dt_node_to_map = pinconf_generic_dt_node_to_map_pin,
    .dt_free_map = pinconf_generic_dt_free_map,
#endif
};
```

The `confops` variable points to the custom `cygpio_pinconf_ops` structure, which contains pointers to callback functions that perform pin config operations. You will only implement the `cygpio_pinconf_set` callback function, which sets the drive modes for all the gpios configured in our Device Tree pin configuration nodes.

```
static const struct pinconf_ops cygpio_pinconf_ops = {
    .pin_config_set = cygpio_pinconf_set,
    .is_generic = true,
};
```

See below the code of the `cygpio_pinconf_set` callback function:

```
/* Configure the Drive Mode Register Settings */
static int cygpio_pinconf_set(struct pinctrl_dev *pctldev, unsigned int pin,
                             unsigned long *configs, unsigned int num_configs)
{
    struct cy8c9520a *cygpio = pinctrl_dev_get_drvdata(pctldev);
    struct i2c_client *client = cygpio->client;
    enum pin_config_param param;
    u32 arg;
    int ret = 0;
    int i;
    u8 offs = 0;
    u8 val = 0;
    u8 port = cypress_get_port(pin);
    u8 pin_offset = cypress_get_offs(pin, port);

    mutex_lock(&cygpio->lock);

    for (i = 0; i < num_configs; i++) {
        param = pinconf_to_config_param(configs[i]);
        arg = pinconf_to_config_argument(configs[i]);

        switch (param) {
            case PIN_CONFIG_BIAS_PULL_UP:
                offs = 0x0;
                break;
            case PIN_CONFIG_BIAS_PULL_DOWN:
                offs = 0x01;
                break;
            case PIN_CONFIG_DRIVE_STRENGTH:
                offs = 0x04;
                break;
            case PIN_CONFIG_BIAS_HIGH_IMPEDANCE:
                offs = 0x06;
                break;
            default:
                dev_err(&client->dev, "Invalid config param %04x\n", param);
                return -ENOTSUPP;
        }

        /* Write to the REG_DRIVE registers of the CY8C9520A device */
        i2c_smbus_write_byte_data(client, REG_PORT_SELECT, port);
```

```

        i2c_smbus_read_byte_data(client, REG_DRIVE_PULLUP + offs);

        val = (u8)(ret | BIT(pin_offset));

        i2c_smbus_write_byte_data(client, REG_DRIVE_PULLUP + offs, val);

    }

    mutex_unlock(&cygpio->lock);
    return ret;
}

```

You will add the following lines in bold to the Device Tree configuration of our cy8c9520a device:

```

cy8c9520a: cy8c9520a@20 {
    compatible = "cy8c9520a";
    reg = <0x20>;
    interrupt-controller;
    #interrupt-cells = <2>;
    gpio-controller;
    #gpio-cells = <2>;

    interrupts = <23 1>;
    interrupt-parent = <&gpio>;

    pinctrl-names = "default";
    pinctrl-0 = <&accel_int_pin &cy8c9520apullups &cy8c9520apulldowns
&cy8c9520adrivestrength>;

    cy8c9520apullups: pinmux1 {
        pins = "gpio0", "gpio1";
        bias-pull-up;
    };

    cy8c9520apulldowns: pinmux2 {
        pins = "gpio2";
        bias-pull-down;
    };

    /* pwm channel */
    cy8c9520adrivestrength: pinmux3 {
        pins = "gpio3";
        drive-strength;
    };
};

```

Inside the gpio node, you will create the accel_int_pin pin configuration node, where the GPIO23 pin is multiplexed as a GPIO signal:

```

accel_int_pin: accel_int_pin {
    brcm,pins = <23>;
    brcm,function = <0>;    /* Input */
    brcm,pull = <0>;        /* none */
}

```

```
};
```

The `pinctrl-x` properties link to a pin configuration node for a given state of the device. The `pinctrl-names` property associates a name to each state. In our driver, we will use only one state, and the name default is used for the `pinctrl-names` property, and it is selected by our device driver without having to make a `pinctrl` function call.

In our DT device node, the `pinctrl-0` property lists several 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. The first pin configuration node applies the pull-up configuration to the `gpio0` and `gpio1` pins (GPort 0, pins 0 and 1). The second pin configuration node applies the pull-down configuration to the `gpio2` pin (GPort 0, pin 2), and finally, the last pin configuration node applies the strong drive configuration to the `gpio3` pin (GPort 0, pin 3). These pin configurations will be written to the CY8C9520A registers by calling the `cygpio_pinconf_set()` callback function previously described.

LAB 7.5 PWM controller driver description

The Linux PWM (Pulse Width Modulation) framework offers an interface that can be used from user space (`sysfs`) and kernel space (API) and allows to:

- Control PWM output(s) such as period, duty cycle and polarity.
- Capture a PWM signal, and report its period and duty cycle.

This section will explain how to implement a PWM controller driver for the CY8C9520A device. As in other frameworks previously explained, there is a main structure that we have to configure and register to the PWM core. The name of this structure is `pwm_chip`, and it will be filled in with a description of the PWM controller, the number of PWM devices provided by the controller, and the chip-specific callback functions which will support the PWM operations. You can see below a code snippet that configures the `pwm_chip` structure:

```
/* Setup of the pwm_chip controller */
cygpio->pwm_chip.dev = &client->dev;
cygpio->pwm_chip.ops = &cy8c9520a_pwm_ops;
cygpio->pwm_chip.base = PWM_BASE_ID;
cygpio->pwm_chip.npwm = NPWM;
```

The `npwm` variable sets the number of PWM channels. The CY8C9520A device has four PWM channels. The `ops` variable points to the `cy8c9520a_pwm_ops` structure, which includes pointers to the PWM chip-specific callback functions that configure, enable and disable the PWM channels of the CY8C9520A device.

```
/* Declare the PWM callback functions */
static const struct pwm_ops cy8c9520a_pwm_ops = {
    .request = cy8c9520a_pwm_request,
```

```

        .config = cy8c9520a_pwm_config,
        .enable = cy8c9520a_pwm_enable,
        .disable = cy8c9520a_pwm_disable,
};

```

The `cy8c9520a_pwm_config` callback function will set up the period and the duty cycle for each PWM channel of the device. The `cy8c9520a_pwm_enable` and `cy8c9520a_pwm_disable` functions will enable/disable each PWM channel of the device. In the listing code of the driver, you can see the full code for these callback functions. These functions will be called from user space by using the `sysfs` method or from kernel space (API) by using a PWM kernel driver. You will use the `sysfs` method during the demonstration section of the driver.

Finally, you will add the following lines in bold to the Device Tree configuration of our `cy8c9520a` device:

```

cy8c9520a: cy8c9520a@20 {
    compatible = "cy8c9520a";
    reg = <0x20>;
    interrupt-controller;
    #interrupt-cells = <2>;
    gpio-controller;
    #gpio-cells = <2>;

    interrupts = <23 1>;
    interrupt-parent = <&gpio>;

    #pwm-cells = <2>;
    pwm0 = <20>; // pwm not supported
    pwm1 = <3>;
    pwm2 = <20>; // pwm not supported
    pwm3 = <2>;

    pinctrl-names = "default";
    pinctrl-0 = <&accel_int_pin &cy8c9520apullups &cy8c9520apulldowns
&cy8c9520adrivestrength>;

    cy8c9520apullups: pinmux1 {
        pins = "gpio0", "gpio1";
        bias-pull-up;
    };

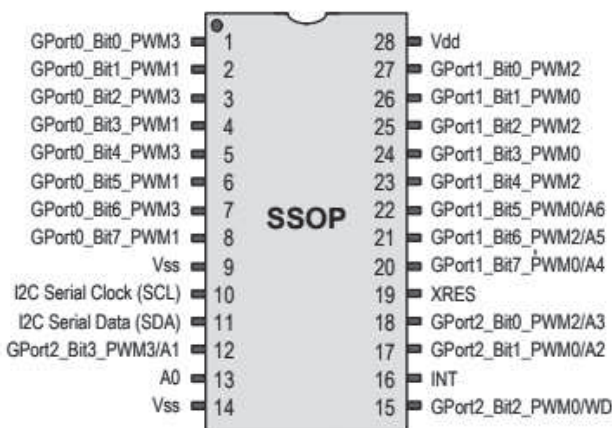
    cy8c9520apulldowns: pinmux2 {
        pins = "gpio2";
        bias-pull-down;
    };

    /* pwm channel */
    cy8c9520adrivestrength: pinmux3 {
        pins = "gpio3";
        drive-strength;
    };
};

```

The `pwmX` property selects the pin of the CY8C9520A device that will be configured as a PWM channel. You will select a pin for every PWM channel (PWM0 to PWM3) of the device. In the following image, extracted from the data-sheet of the CY8C9520A device, you can see which PWM channel is associated with each port pin of the device. In the Device Tree, you will set the `pwm1` channel to the Bit 3 (`gpio3`) of the GPort0 and the `pwm3` channel to the bit 2 (`gpio2`) of the GPort0. If a PWM channel is not used, you will set its `pwmX` property to a value of 20. This configuration is just an example, you can of course add your own configuration.

Figure 2. CY8C9520A 28-Pin Device



You will recover the values of the `pwmX` properties by using the `device_property_read_u32()` function, as shown in the following code snippet:

```
/* parse the DT to get the pwm-pin mapping */
for (i = 0; i < NPWM; i++) {
    ret = device_property_read_u32(&client->dev, name[i], &tmp);
    if (!ret)
        cygpio->pwm_number[i] = tmp;
    else
        goto err;
};
```

Create a new `linux_5.4_CY8C9520A_pwm_pinctrl` folder inside the `linux_5.4_rpi3_drivers` folder:

```
~/linux_5.4_rpi3_drivers$ mkdir linux_5.4_CY8C9520A_pwm_pinctrl
```

Create a new `CY8C9520A_pwm_pinctrl.c` file and a Makefile file in the `linux_5.4_CY8C9520A_pwm_pinctrl` folder, and add `CY8C9520A_pwm_pinctrl.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_pwm_pinctrl$ make
~/linux_5.4_rpi3_drivers/linux_5.4_CY8C9520A_pwm_pinctrl$ make deploy
```

Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 7-7: CY8C9520A_pwm_pinctrl.c

```
#include <linux/i2c.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/gpio/driver.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/pwm.h>
#include <linux/slab.h>
#include <linux/pinctrl/pinctrl.h>
#include <linux/pinctrl/pinconf.h>
#include <linux/pinctrl/pinconf-generic.h>

#define DRV_NAME "cy8c9520a"

/* cy8c9520a settings */
#define NGPIO 20
#define DEVID_CY8C9520A 0x20
#define NPORTS 3
#define NPWM 4
#define PWM_MAX_PERIOD 0xff
#define PWM_BASE_ID 0
#define PWM_CLK 0x00 /* see resulting PWM_TCLK_NS */
#define PWM_TCLK_NS 31250 /* 32kHz */
#define PWM_UNUSED 20

/* Register offset */
#define REG_INPUT_PORT0 0x00
#define REG_OUTPUT_PORT0 0x08
#define REG_INTR_STAT_PORT0 0x10
#define REG_PORT_SELECT 0x18
#define REG_INTR_MASK 0x19
#define REG_PIN_DIR 0x1c
#define REG_DRIVE_PULLUP 0x1d
#define REG_DRIVE_PULLDOWN 0x1e
```



```

#define REG_DEVID_STAT                0x2e

/* Register PWM */
#define REG_SELECT_PWM                0x1a
#define REG_PWM_SELECT                0x28
#define REG_PWM_CLK                   0x29
#define REG_PWM_PERIOD                0x2a
#define REG_PWM_PULSE_W               0x2b

/* Definition of the global structure for the driver */
struct cy8c9520a {
    struct i2c_client      *client;
    struct gpio_chip       gpio_chip;
    struct irq_chip        irq_chip;
    struct pwm_chip        pwm_chip;
    struct gpio_desc       *gpio;
    int                    irq;
    struct mutex           lock;
    /* protect serialized access to the interrupt controller bus */
    struct mutex           irq_lock;
    /* cached output registers */
    u8                     outreg_cache[NPORTS];
    /* cached IRQ mask */
    u8                     irq_mask_cache[NPORTS];
    /* IRQ mask to be applied */
    u8                     irq_mask[NPORTS];
    int                    pwm_number[NPWM];

    struct pinctrl_dev      *pctldev;
    struct pinctrl_desc     pinctrl_desc;
};

/* Per-port GPIO offset */
static const u8 cy8c9520a_port_offs[] = {
    0,
    8,
    16,
};

static const struct pinctrl_pin_desc cy8c9520a_pins[] = {
    PINCTRL_PIN(0, "gpio0"),
    PINCTRL_PIN(1, "gpio1"),
    PINCTRL_PIN(2, "gpio2"),
    PINCTRL_PIN(3, "gpio3"),
    PINCTRL_PIN(4, "gpio4"),
    PINCTRL_PIN(5, "gpio5"),
    PINCTRL_PIN(6, "gpio6"),
    PINCTRL_PIN(7, "gpio7"),
    PINCTRL_PIN(8, "gpio8"),
    PINCTRL_PIN(9, "gpio9"),
    PINCTRL_PIN(10, "gpio10"),
    PINCTRL_PIN(11, "gpio11"),
    PINCTRL_PIN(12, "gpio12"),
    PINCTRL_PIN(13, "gpio13"),
};

```

```

    PINCTRL_PIN(14, "gpio14"),
    PINCTRL_PIN(15, "gpio15"),
    PINCTRL_PIN(16, "gpio16"),
    PINCTRL_PIN(17, "gpio17"),
    PINCTRL_PIN(18, "gpio18"),
    PINCTRL_PIN(19, "gpio19"),
};

/* Return the port of the gpio */
static inline u8 cypress_get_port(unsigned int gpio)
{
    u8 i = 0;
    for (i = 0; i < sizeof(cy8c9520a_port_offs) - 1; i++) {
        if (! (gpio / cy8c9520a_port_offs[i + 1]))
            break;
    }
    return i;
}

/* Get the gpio offset inside its respective port */
static inline u8 cypress_get_offs(unsigned gpio, u8 port)
{
    return gpio - cy8c9520a_port_offs[port];
}

static int cygpio_pinctrl_get_groups_count(struct pinctrl_dev *pctldev)
{
    return 0;
}

static const char *cygpio_pinctrl_get_group_name(struct pinctrl_dev *pctldev,
                                                  unsigned int group)
{
    return NULL;
}

static int cygpio_pinctrl_get_group_pins(struct pinctrl_dev *pctldev,
                                         unsigned int group,
                                         const unsigned int **pins,
                                         unsigned int *num_pins)
{
    return -ENOTSUPP;
}

/*
 * Global pin control operations implemented by
 * pin controller drivers.
 * pinconf_generic_dt_node_to_map_pin function
 * will parse a device tree "pin configuration node", and create
 * mapping table entries for it
 */
static const struct pinctrl_ops cygpio_pinctrl_ops = {
    .get_groups_count = cygpio_pinctrl_get_groups_count,
    .get_group_name = cygpio_pinctrl_get_group_name,

```

```

    .get_group_pins = cygpio_pinctrl_get_group_pins,
#ifdef CONFIG_OF
    .dt_node_to_map = pinconf_generic_dt_node_to_map_pin,
    .dt_free_map = pinconf_generic_dt_free_map,
#endif
};

/* Configure the Drive Mode Register Settings */
static int cygpio_pinconf_set(struct pinctrl_dev *pctldev, unsigned int pin,
                             unsigned long *configs, unsigned int num_configs)
{
    struct cy8c9520a *cygpio = pinctrl_dev_get_drvdata(pctldev);
    struct i2c_client *client = cygpio->client;
    enum pin_config_param param;
    u32 arg;
    int ret = 0;
    int i;
    u8 offs = 0;
    u8 val = 0;
    u8 port = cypress_get_port(pin);
    u8 pin_offset = cypress_get_offs(pin, port);

    dev_err(&client->dev, "cygpio_pinconf_set function is called\n");

    mutex_lock(&cygpio->lock);

    for (i = 0; i < num_configs; i++) {
        param = pinconf_to_config_param(configs[i]);
        arg = pinconf_to_config_argument(configs[i]);

        switch (param) {
            case PIN_CONFIG_BIAS_PULL_UP:
                offs = 0x0;
                dev_info(&client->dev,
                        "The pin %d drive mode is PIN_CONFIG_BIAS_PULL_UP\n", pin);
                break;
            case PIN_CONFIG_BIAS_PULL_DOWN:
                offs = 0x01;
                dev_info(&client->dev,
                        "The pin %d drive mode is PIN_CONFIG_BIAS_PULL_DOWN\n", pin);
                break;
            case PIN_CONFIG_DRIVE_STRENGTH:
                offs = 0x04;
                dev_info(&client->dev,
                        "The pin %d drive mode is PIN_CONFIG_DRIVE_STRENGTH\n", pin);
                break;
            case PIN_CONFIG_BIAS_HIGH_IMPEDANCE:
                offs = 0x06;
                dev_info(&client->dev,
                        "The pin %d drive mode is PIN_CONFIG_BIAS_HIGH_IMPEDANCE\n", pin);
                break;
            default:
                dev_err(&client->dev, "Invalid config param %04x\n", param);
                return -ENOTSUPP;
        }
    }
}

```

```

    }

    ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, port);
    if (ret < 0) {
        dev_err(&client->dev, "can't select port %u\n", port);
        goto end;
    }

    ret = i2c_smbus_read_byte_data(client, REG_DRIVE_PULLUP + offs);
    if (ret < 0) {
        dev_err(&client->dev, "can't read pin direction\n");
        goto end;
    }

    val = (u8)(ret | BIT(pin_offset));

    ret = i2c_smbus_write_byte_data(client, REG_DRIVE_PULLUP + offs, val);
    if (ret < 0) {
        dev_err(&client->dev, "can't set drive mode port %u\n", port);
        goto end;
    }

}

end:
    mutex_unlock(&cygpio->lock);
    return ret;
}

/*
 * pin config operations implemented by
 * pin configuration capable drivers
 * pin_config_set: configure an individual pin
 */
static const struct pinconf_ops cygpio_pinconf_ops = {
    .pin_config_set = cygpio_pinconf_set,
    .is_generic = true,
};

/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the REG_INPUT_PORT register
 */
static int cy8c9520a_gpio_get(struct gpio_chip *chip, unsigned int gpio)
{
    int ret;
    u8 port, in_reg;

    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    dev_info(chip->parent, "cy8c9520a_gpio_get function is called\n");

    /* Get the input port address address (in_reg) for the GPIO */

```

```

port = cypress_get_port(gpio);
in_reg = REG_INPUT_PORT0 + port;

dev_info(chip->parent, "the in_reg address is %u\n", in_reg);

mutex_lock(&cygpio->lock);

ret = i2c_smbus_read_byte_data(cygpio->client, in_reg);
if (ret < 0) {
    dev_err(chip->parent, "can't read input port %u\n", in_reg);
}

dev_info(chip->parent, "cy8c9520a_gpio_get function with %d value is returned\n", ret);

mutex_unlock(&cygpio->lock);

/*
 * Check the status of the GPIO in its input port register
 * and return it. If expression is not 0 returns 1
 */
return !(ret & BIT(cypress_get_offs(gpio, port)));
}

/*
 * struct gpio_chip set callback function.
 * It sets the output value of the GPIO line in
 * GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the REG_OUTPUT_PORT register
 */
static void cy8c9520a_gpio_set(struct gpio_chip *chip, unsigned int gpio, int val)
{
    int ret;
    u8 port, out_reg;
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    dev_info(chip->parent, "cy8c9520a_gpio_set_value func with %d value is called\n", val);

    /* Get the output port address address (out_reg) for the GPIO */
    port = cypress_get_port(gpio);
    out_reg = REG_OUTPUT_PORT0 + port;

    mutex_lock(&cygpio->lock);

    /*
     * If val is 1, gpio output level is high.
     * If val is 0, gpio output level is low.
     * The output registers were previously cached in cy8c9520a_setup()
     */
    if (val) {
        cygpio->outreg_cache[port] |= BIT(cypress_get_offs(gpio, port));
    } else {
        cygpio->outreg_cache[port] &= ~BIT(cypress_get_offs(gpio, port));
    }
}

```

```

    ret = i2c_smbus_write_byte_data(cygpio->client, out_reg, cygpio->outreg_cache[port]);
    if (ret < 0) {
        dev_err(chip->parent, "can't write output port %u\n", port);
    }

    mutex_unlock(&cygpio->lock);
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO as an output writing to
 * the REG_PIN_DIR register of the selected port
 */
static int cy8c9520a_gpio_direction_output(struct gpio_chip *chip, unsigned int gpio, int val)
{
    int ret;
    u8 pins, port;

    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    /* gets the port number of the gpio */
    port = cypress_get_port(gpio);

    dev_info(chip->parent, "cy8c9520a_gpio_direction output is called\n");

    mutex_lock(&cygpio->lock);

    /* Select the port where we want to config the GPIO as output */
    ret = i2c_smbus_write_byte_data(cygpio->client, REG_PORT_SELECT, port);
    if (ret < 0) {
        dev_err(chip->parent, "can't select port %u\n", port);
        goto err;
    }

    ret = i2c_smbus_read_byte_data(cygpio->client, REG_PIN_DIR);
    if (ret < 0) {
        dev_err(chip->parent, "can't read pin direction\n");
        goto err;
    }

    /* Simply transform int to u8 */
    pins = (u8)ret & 0xff;

    /* Add the direction of the new pin. Set 1 if input and set 0 is output */
    pins &= ~BIT(cypress_get_offs(gpio, port));

    ret = i2c_smbus_write_byte_data(cygpio->client, REG_PIN_DIR, pins);
    if (ret < 0) {
        dev_err(chip->parent, "can't write pin direction\n");
    }

err:
    mutex_unlock(&cygpio->lock);
    cy8c9520a_gpio_set(chip, gpio, val);
}

```

```

    return ret;
}

/*
 * struct gpio_chip direction_input callback function.
 * It configures the GPIO as an input writing to
 * the REG_PIN_DIR register of the selected port
 */
static int cy8c9520a_gpio_direction_input(struct gpio_chip *chip, unsigned int gpio)
{
    int ret;
    u8 pins, port;

    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    /* Gets the port number of the gpio */
    port = cypress_get_port(gpio);

    dev_info(chip->parent, "cy8c9520a_gpio_direction input is called\n");

    mutex_lock(&cygpio->lock);

    /* Select the port where we want to config the GPIO as input */
    ret = i2c_smbus_write_byte_data(cygpio->client, REG_PORT_SELECT, port);
    if (ret < 0) {
        dev_err(chip->parent, "can't select port %u\n", port);
        goto err;
    }

    ret = i2c_smbus_read_byte_data(cygpio->client, REG_PIN_DIR);
    if (ret < 0) {
        dev_err(chip->parent, "can't read pin direction\n");
        goto err;
    }

    /* Simply transform int to u8 */
    pins = (u8)ret & 0xff;

    /*
     * Add the direction of the new pin.
     * Set 1 if input (out == 0) and set 0 is output (out == 1)
     */
    pins |= BIT(cypress_get_offs(gpio, port));

    ret = i2c_smbus_write_byte_data(cygpio->client, REG_PIN_DIR, pins);
    if (ret < 0) {
        dev_err(chip->parent, "can't write pin direction\n");
        goto err;
    }

err:
    mutex_unlock(&cygpio->lock);
    return ret;
}

```

```

/* Function to lock access to slow bus (i2c) chips */
static void cy8c9520a_irq_bus_lock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    dev_info(chip->parent, "cy8c9520a_irq_bus_lock is called\n");
    mutex_lock(&cygpio->irq_lock);
}

/*
 * Function to sync and unlock slow bus (i2c) chips.
 * REG_INTR_MASK register is accessed via I2C
 * Write 0 to the interrupt mask register line to
 * activate the interrupt on the GPIO
 */
static void cy8c9520a_irq_bus_sync_unlock(struct irq_data *d)
{
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    int ret, i;
    unsigned int gpio;
    u8 port;
    dev_info(chip->parent, "cy8c9520a_irq_bus_sync_unlock is called\n");
    gpio = d->hwirq;
    port = cypress_get_port(gpio);

    /* irq_mask_cache stores the last value of irq_mask for each port */
    for (i = 0; i < NPORTS; i++) {
        /*
         * Check if some of the bits have changed from the last cached value.
         * irq_mask registers were initialized in cy8c9520a_irq_setup()
         */
        if (cygpio->irq_mask_cache[i] ^ cygpio->irq_mask[i]) {
            dev_info(chip->parent, "gpio %u is unmasked\n", gpio);
            cygpio->irq_mask_cache[i] = cygpio->irq_mask[i];
            ret = i2c_smbus_write_byte_data(cygpio->client,
                                           REG_PORT_SELECT, i);

            if (ret < 0) {
                dev_err(chip->parent, "can't select port %u\n", port);
                goto err;
            }

            /* Enable the interrupt for the GPIO unmasked */
            ret = i2c_smbus_write_byte_data(cygpio->client, REG_INTR_MASK,
                                           cygpio->irq_mask[i]);

            if (ret < 0) {
                dev_err(chip->parent,
                    "can't write int mask on port %u\n", port);
                goto err;
            }

            ret = i2c_smbus_read_byte_data(cygpio->client, REG_INTR_MASK);
            dev_info(chip->parent, "the REG_INTR_MASK value is %d\n", ret);
        }
    }
}

```



```

    }
}

err:
    mutex_unlock(&cygpio->irq_lock);
}

/*
 * Mask (disable) the GPIO interrupt.
 * In the initial setup all the INT lines are masked
 */
static void cy8c9520a_irq_mask(struct irq_data *d)
{
    u8 port;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    unsigned gpio = d->hwirq;
    port = cypress_get_port(gpio);
    dev_info(chip->parent, "cy8c9520a_irq_mask is called\n");

    cygpio->irq_mask[port] |= BIT(cypress_get_offs(gpio, port));
}

/*
 * Unmask (enable) the GPIO interrupt.
 * In the initial setup all the INT lines are masked
 */
static void cy8c9520a_irq_unmask(struct irq_data *d)
{
    u8 port;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);
    unsigned gpio = d->hwirq;
    port = cypress_get_port(gpio);
    dev_info(chip->parent, "cy8c9520a_irq_unmask is called\n");

    cygpio->irq_mask[port] &= ~BIT(cypress_get_offs(gpio, port));
}

/* Set the flow type (IRQ_TYPE_LEVEL/etc.) of the IRQ */
static int cy8c9520a_irq_set_type(struct irq_data *d, unsigned int type)
{
    int ret = 0;
    struct gpio_chip *chip = irq_data_get_irq_chip_data(d);
    struct cy8c9520a *cygpio = gpiochip_get_data(chip);

    dev_info(chip->parent, "cy8c9520a_irq_set_type is called\n");

    if ((type != IRQ_TYPE_EDGE_BOTH) && (type != IRQ_TYPE_EDGE_FALLING)) {
        dev_err(&cygpio->client->dev, "irq %d: unsupported type %d\n", d->irq, type);
        ret = -EINVAL;
        goto err;
    }
}

```

```

err:
    return ret;
}

/* Initialization of the irq_chip structure with callback functions */
static struct irq_chip cy8c9520a_irq_chip = {
    .name                = "cy8c9520a-irq",
    .irq_mask            = cy8c9520a_irq_mask,
    .irq_unmask         = cy8c9520a_irq_unmask,
    .irq_bus_lock       = cy8c9520a_irq_bus_lock,
    .irq_bus_sync_unlock = cy8c9520a_irq_bus_sync_unlock,
    .irq_set_type       = cy8c9520a_irq_set_type,
};

/*
 * Interrupt handler for the cy8c9520a. It is called when
 * there is a rising or falling edge in the unmasked GPIO
 */
static irqreturn_t cy8c9520a_irq_handler(int irq, void *devid)
{
    struct cy8c9520a *cygpio = devid;
    u8 stat[NPORTS], pending;
    unsigned port, gpio, gpio_irq;
    int ret;

    pr_info ("the interrupt ISR has been entered\n");

    /*
     * Store in stat and clear (to enable ints)
     * the three interrupt status registers by reading them
     */
    ret = i2c_smbus_read_i2c_block_data(cygpio->client, REG_INTR_STAT_PORT0, NPORTS, stat);
    if (ret < 0) {
        memset(stat, 0, sizeof(stat));
    }

    ret = IRQ_NONE;

    for (port = 0; port < NPORTS; port++) {
        mutex_lock(&cygpio->irq_lock);

        /*
         * In every port, check the GPIOs that have their INTs unmasked
         * and whose bits have been enabled in their REG_INTR_STAT_PORT
         * register due to an interrupt in the GPIO, then store the new
         * value in the pending register
         */
        pending = stat[port] & (~cygpio->irq_mask[port]);
        mutex_unlock(&cygpio->irq_lock);

        /* Launch the ISRs of all the gpios that requested an interrupt */
        while (pending) {
            ret = IRQ_HANDLED;

```

```

        /* get the first gpio that has got an int */
        gpio = __ffs(pending);

        /* clears the gpio in the pending register */
        pending &= ~BIT(gpio);

        /* gets the int number associated to this gpio */
        gpio_irq = cy8c9520a_port_offs[port] + gpio;

        /* launch the ISR of the GPIO child driver */
        handle_nested_irq(irq_find_mapping(cygpio->gpio_chip.irq.domain, gpio_irq));
    }
}

return ret;
}

/*
 * Select the period and the duty cycle of the PWM signal (in nanoseconds):
 * echo 100000 > pwm1/period
 * echo 50000 > pwm1/duty_cycle
 */
static int cy8c9520a_pwm_config(struct pwm_chip *chip, struct pwm_device *pwm,
                               int duty_ns, int period_ns)
{
    int ret;
    int period = 0, duty = 0;

    struct cy8c9520a *cygpio = container_of(chip, struct cy8c9520a, pwm_chip);
    struct i2c_client *client = cygpio->client;

    dev_info(&client->dev, "cy8c9520a_pwm_config is called\n");

    if (pwm->pwm > NPWM) {
        return -EINVAL;
    }

    period = period_ns / PWM_TCLK_NS;
    duty = duty_ns / PWM_TCLK_NS;

    /*
     * Check period's upper bound. Note the duty cycle is already sanity
     * checked by the PWM framework.
     */
    if (period > PWM_MAX_PERIOD) {
        dev_err(&client->dev, "period must be within [0-%d]ns\n",
                PWM_MAX_PERIOD * PWM_TCLK_NS);
        return -EINVAL;
    }

    mutex_lock(&cygpio->lock);

    /*
     * Select the pwm number (from 0 to 3)

```

```

    * to set the period and the duty for the enabled pwm pins
    */
    ret = i2c_smbus_write_byte_data(client, REG_PWM_SELECT, (u8)pwm->pwm);
    if (ret < 0) {
        dev_err(&client->dev, "can't write to REG_PWM_SELECT\n");
        goto end;
    }

    ret = i2c_smbus_write_byte_data(client, REG_PWM_PERIOD, (u8)period);
    if (ret < 0) {
        dev_err(&client->dev, "can't write to REG_PWM_PERIOD\n");
        goto end;
    }

    ret = i2c_smbus_write_byte_data(client, REG_PWM_PULSE_W, (u8)duty);
    if (ret < 0) {
        dev_err(&client->dev, "can't write to REG_PWM_PULSE_W\n");
        goto end;
    }

end:
    mutex_unlock(&cygpio->lock);

    return ret;
}

/*
 * Enable the PWM signal:
 * echo 1 > pwm1/enable
 */
static int cy8c9520a_pwm_enable(struct pwm_chip *chip, struct pwm_device *pwm)
{
    int ret, gpio, port, pin;
    u8 out_reg, val;

    struct cy8c9520a *cygpio = container_of(chip, struct cy8c9520a, pwm_chip);
    struct i2c_client *client = cygpio->client;

    dev_info(&client->dev, "cy8c9520a_pwm_enable is called\n");

    if (pwm->pwm > NPWM) {
        return -EINVAL;
    }

    /*
     * Get the pin configured as pwm in the Device Tree
     * for this pwm port (pwm_device)
     */
    gpio = cygpio->pwm_number[pwm->pwm];
    port = cypress_get_port(gpio);
    pin = cypress_get_offs(gpio, port);
    out_reg = REG_OUTPUT_PORT0 + port;

    /*

```

```

    * Set pin as output driving high and select the port
    * where the pwm will be set
    */
ret = cy8c9520a_gpio_direction_output(&cygpio->gpio_chip, gpio, 1);
if (val < 0) {
    dev_err(&client->dev, "can't set pwm%u as output\n", pwm->pwm);
    return ret;
}

mutex_lock(&cygpio->lock);

/* Enable PWM pin in the selected port */
val = i2c_smbus_read_byte_data(client, REG_SELECT_PWM);
if (val < 0) {
    dev_err(&client->dev, "can't read REG_SELECT_PWM\n");
    ret = val;
    goto end;
}
val |= BIT((u8)pin);
ret = i2c_smbus_write_byte_data(client, REG_SELECT_PWM, val);
if (ret < 0) {
    dev_err(&client->dev, "can't write to SELECT_PWM\n");
    goto end;
}

end:
mutex_unlock(&cygpio->lock);

return ret;
}

/*
 * Disable the PWM signal:
 * echo 0 > pwm1/enable
 */
static void cy8c9520a_pwm_disable(struct pwm_chip *chip, struct pwm_device *pwm)
{
    int ret, gpio, port, pin;
    u8 val;

    struct cy8c9520a *cygpio = container_of(chip, struct cy8c9520a, pwm_chip);
    struct i2c_client *client = cygpio->client;

    dev_info(&client->dev, "cy8c9520a_pwm_disable is called\n");

    if (pwm->pwm > NPWM) {
        return;
    }

    gpio = cygpio->pwm_number[pwm->pwm];
    if (PWM_UNUSED == gpio) {
        dev_err(&client->dev, "pwm%d is unused\n", pwm->pwm);
        return;
    }
}

```

```

port = cypress_get_port(gpio);
pin = cypress_get_offs(gpio, port);

mutex_lock(&cygpio->lock);

/* Disable PWM */
val = i2c_smbus_read_byte_data(client, REG_SELECT_PWM);
if (val < 0) {
    dev_err(&client->dev, "can't read REG_SELECT_PWM\n");
    goto end;
}
val &= ~BIT((u8)pin);
ret = i2c_smbus_write_byte_data(client, REG_SELECT_PWM, val);
if (ret < 0) {
    dev_err(&client->dev, "can't write to SELECT_PWM\n");
}

end:
mutex_unlock(&cygpio->lock);

return;
}

/*
 * Request the PWM device:
 * echo 0 > export
 */
static int cy8c9520a_pwm_request(struct pwm_chip *chip, struct pwm_device *pwm)
{
    int gpio = 0;
    struct cy8c9520a *cygpio = container_of(chip, struct cy8c9520a, pwm_chip);
    struct i2c_client *client = cygpio->client;

    dev_info(&client->dev, "cy8c9520a_pwm_request is called\n");

    if (pwm->pwm > NPWM) {
        return -EINVAL;
    }

    gpio = cygpio->pwm_number[pwm->pwm];
    if (PWM_UNUSED == gpio) {
        dev_err(&client->dev, "pwm%d unavailable\n", pwm->pwm);
        return -EINVAL;
    }

    return 0;
}

/* Declare the PWM callback functions */
static const struct pwm_ops cy8c9520a_pwm_ops = {
    .request = cy8c9520a_pwm_request,
    .config = cy8c9520a_pwm_config,
    .enable = cy8c9520a_pwm_enable,

```

```

        .disable = cy8c9520a_pwm_disable,
    };

/* Initial setup for the cy8c9520a */
static int cy8c9520a_setup(struct cy8c9520a *cygpio)
{
    int ret, i;
    struct i2c_client *client = cygpio->client;

    /* Disable PWM, set all GPIOs as input */
    for (i = 0; i < NPORTS; i++) {
        ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
        if (ret < 0) {
            dev_err(&client->dev, "can't select port %u\n", i);
            goto end;
        }

        ret = i2c_smbus_write_byte_data(client, REG_SELECT_PWM, 0x00);
        if (ret < 0) {
            dev_err(&client->dev, "can't write to SELECT_PWM\n");
            goto end;
        }

        ret = i2c_smbus_write_byte_data(client, REG_PIN_DIR, 0xff);
        if (ret < 0) {
            dev_err(&client->dev, "can't write to PIN_DIR\n");
            goto end;
        }
    }

    /* Cache the output registers (Output Port 0, Output Port 1, Output Port 2) */
    ret = i2c_smbus_read_i2c_block_data(client, REG_OUTPUT_PORT0,
                                        sizeof(cygpio->outreg_cache),
                                        cygpio->outreg_cache);

    if (ret < 0) {
        dev_err(&client->dev, "can't cache output registers\n");
        goto end;
    }

    /* Set default PWM clock source */
    for (i = 0; i < NPWM; i++) {
        ret = i2c_smbus_write_byte_data(client, REG_PWM_SELECT, i);
        if (ret < 0) {
            dev_err(&client->dev, "can't select pwm %u\n", i);
            goto end;
        }

        ret = i2c_smbus_write_byte_data(client, REG_PWM_CLK, PWM_CLK);
        if (ret < 0) {
            dev_err(&client->dev, "can't write to REG_PWM_CLK\n");
            goto end;
        }
    }
}

```

```

    dev_info(&client->dev, "the cy8c9520a_setup is done\n");

end:
    return ret;
}

/* Interrupt setup for the cy8c9520a device */
static int cy8c9520a_irq_setup(struct cy8c9520a *cygpio)
{
    struct gpio_irq_chip *girq;
    struct i2c_client *client = cygpio->client;
    u8 dummy[NPORTS];
    int ret, i;

    mutex_init(&cygpio->irq_lock);

    dev_info(&client->dev, "the cy8c9520a_irq_setup function is entered\n");

    /*
     * Clear interrupt state registers by reading the three registers:
     * Interrupt Status Port0, Interrupt Status Port1, Interrupt Status Port2,
     * and store the values in a dummy array
     */
    ret = i2c_smbus_read_i2c_block_data(client, REG_INTR_STAT_PORT0, NPORTS, dummy);
    if (ret < 0) {
        dev_err(&client->dev, "couldn't clear int status\n");
        goto err;
    }

    dev_info(&client->dev, "the interrupt state registers are cleared\n");

    /*
     * Initialize Interrupt Mask Port Register (19h) for each port.
     * Disable the activation of the INT lines. Each 1 in this
     * register masks (disables) the INT for the corresponding GPIO
     */
    memset(cygpio->irq_mask_cache, 0xff, sizeof(cygpio->irq_mask_cache));
    memset(cygpio->irq_mask, 0xff, sizeof(cygpio->irq_mask));

    /* Disable interrupts in all the gpio lines */
    for (i = 0; i < NPORTS; i++) {
        ret = i2c_smbus_write_byte_data(client, REG_PORT_SELECT, i);
        if (ret < 0) {
            dev_err(&client->dev, "can't select port %u\n", i);
            goto err;
        }

        ret = i2c_smbus_write_byte_data(client, REG_INTR_MASK, cygpio->irq_mask[i]);
        if (ret < 0) {
            dev_err(&client->dev, "can't write int mask on port %u\n", i);
            goto err;
        }
    }
}

```



```

dev_info(&client->dev, "the interrupt mask port registers are set\n");

/*
 * Request interrupt on a GPIO pin of the external processor.
 * This processor pin is connected to the INT pin of the cy8c9520a
 */
ret = devm_request_threaded_irq(&client->dev, client->irq, NULL,
                                cy8c9520a_irq_handler,
                                IRQF_ONESHOT | IRQF_TRIGGER_HIGH,
                                dev_name(&client->dev), cygpio);

if (ret) {
    dev_err(&client->dev, "failed to request irq %d\n", cygpio->irq);
    return ret;
}

/*
 * Set up a nested irq handler for a gpio_chip from a parent IRQ.
 * You can now request interrupts from GPIO child drivers nested
 * to the cy8c9520a driver
 */
girq = &cygpio->gpio_chip.irq;
girq->chip = &cy8c9520a_irq_chip;
girq->parent_handler = NULL;
girq->num_parents = 0;
girq->parents = NULL;
girq->default_type = IRQ_TYPE_NONE;
girq->handler = handle_simple_irq;
girq->threaded = true;

dev_info(&client->dev, "the interrupt setup is done\n");

return 0;
err:
mutex_destroy(&cygpio->irq_lock);
return ret;
}

/*
 * Initialize the cy8c9520a gpio controller (struct gpio_chip)
 * and register it to the kernel
 */
static void cy8c9520a_gpio_init(struct cy8c9520a *cygpio)
{
    struct gpio_chip *gpiochip = &cygpio->gpio_chip;

    gpiochip->label = cygpio->client->name;
    gpiochip->base = -1;
    gpiochip->ngpio = NGPIO;
    gpiochip->parent = &cygpio->client->dev;
    gpiochip->of_node = gpiochip->parent->of_node;
    gpiochip->can_sleep = true;
    gpiochip->direction_input = cy8c9520a_gpio_direction_input;
    gpiochip->direction_output = cy8c9520a_gpio_direction_output;
    gpiochip->get = cy8c9520a_gpio_get;

```

```

    gpiochip->set = cy8c9520a_gpio_set;
    gpiochip->owner = THIS_MODULE;
}

static int cy8c9520a_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    struct cy8c9520a *cygpio;
    int ret = 0;
    int i;
    unsigned int dev_id, tmp;
    static const char * const name[] = { "pwm0", "pwm1", "pwm2", "pwm3" };

    dev_info(&client->dev, "cy8c9520a_probe() function is called\n");

    if (!i2c_check_functionality(client->adapter,
                                I2C_FUNC_SMBUS_I2C_BLOCK | I2C_FUNC_SMBUS_BYTE_DATA)) {
        dev_err(&client->dev, "SMBUS Byte/Block unsupported\n");
        return -EIO;
    }

    /* Allocate a global private structure for the device */
    cygpio = devm_kzalloc(&client->dev, sizeof(*cygpio), GFP_KERNEL);
    if (!cygpio) {
        dev_err(&client->dev, "failed to alloc memory\n");
        return -ENOMEM;
    }

    cygpio->client = client;

    mutex_init(&cygpio->lock);

    /* Whoami */
    dev_id = i2c_smbus_read_byte_data(client, REG_DEVID_STAT);
    if (dev_id < 0) {
        dev_err(&client->dev, "can't read device ID\n");
        ret = dev_id;
        goto err;
    }
    dev_info(&client->dev, "dev_id=0x%x\n", dev_id & 0xff);

    /* Parse the DT to get the pwm-pin mapping */
    for (i = 0; i < NPWM; i++) {
        ret = device_property_read_u32(&client->dev, name[i], &tmp);
        if (!ret)
            cygpio->pwm_number[i] = tmp;
        else
            goto err;
    };

    /* Initial setup for the cy8c9520a */
    ret = cy8c9520a_setup(cygpio);
    if (ret < 0) {
        goto err;
    }
}

```

```

dev_info(&client->dev, "the initial setup for the cy8c9520a is done\n");

/* Initialize the cy8c9520a gpio controller */
cy8c9520a_gpio_init(cygpio);

dev_info(&client->dev, "the setup for the cy8c9520a gpio controller done\n");

/* Interrupt setup for the cy8c9520a */
ret = cy8c9520a_irq_setup(cygpio);
if (ret) {
    goto err;
}

dev_info(&client->dev, "the interrupt setup for the cy8c9520a is done\n");

ret = devm_gpiochip_add_data(&client->dev, &cygpio->gpio_chip, cygpio);
if (ret) {
    goto err;
}

/* Setup of the pwm_chip controller */
cygpio->pwm_chip.dev = &client->dev;
cygpio->pwm_chip.ops = &cy8c9520a_pwm_ops;
cygpio->pwm_chip.base = PWM_BASE_ID;
cygpio->pwm_chip.npwm = NPWM;

ret = pwmchip_add(&cygpio->pwm_chip);
if (ret) {
    dev_err(&client->dev, "pwmchip_add failed %d\n", ret);
    goto err;
}

dev_info(&client->dev, "the setup for the cy8c9520a pwm_chip controller is done\n");

/* Setup of the pinctrl descriptor */
cygpio->pinctrl_desc.name = "cy8c9520a-pinctrl";
cygpio->pinctrl_desc.pctlops = &cygpio_pinctrl_ops;
cygpio->pinctrl_desc.confops = &cygpio_pinconf_ops;
cygpio->pinctrl_desc.npins = cygpio->gpio_chip.ngpio;

cygpio->pinctrl_desc.pins = cy8c9520a_pins;
cygpio->pinctrl_desc.owner = THIS_MODULE;

cygpio->pctldev = devm_pinctrl_register(&client->dev, &cygpio->pinctrl_desc, cygpio);
if (IS_ERR(cygpio->pctldev)) {
    ret = PTR_ERR(cygpio->pctldev);
    goto err;
}

dev_info(&client->dev, "the setup for the cy8c9520a pinctrl descriptor is done\n");

/* Link the I2C device with the cygpio device */
i2c_set_clientdata(client, cygpio);

```

```

err:
    mutex_destroy(&cygpio->lock);

    return ret;
}

static int cy8c9520a_remove(struct i2c_client *client)
{
    struct cy8c9520a *cygpio = i2c_get_clientdata(client);
    dev_info(&client->dev, "cy8c9520a_remove() function is called\n");
    return pwmchip_remove(&cygpio->pwm_chip);
}

static const struct of_device_id my_of_ids[] = {
    { .compatible = "cy8c9520a"},
    {}
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static const struct i2c_device_id cy8c9520a_id[] = {
    {DRV_NAME, 0},
    {}
};
MODULE_DEVICE_TABLE(i2c, cy8c9520a_id);

static struct i2c_driver cy8c9520a_driver = {
    .driver = {
        .name = DRV_NAME,
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    },
    .probe = cy8c9520a_probe,
    .remove = cy8c9520a_remove,
    .id_table = cy8c9520a_id,
};
module_i2c_driver(cy8c9520a_driver);

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a driver that controls the cy8c9520a I2C GPIO expander");

```

LAB 7.5 driver demonstration

Load the CY8C9520A_pwm_pinctrl.ko module:

```
root@raspberrypi:/home/pi# insmod CY8C9520A_pwm_pinctrl.ko
cy8c9520a 1-0020: cy8c9520a_probe() function is called
cy8c9520a 1-0020: dev_id=0x20
cy8c9520a 1-0020: the cy8c9520a_setup is done
cy8c9520a 1-0020: the initial setup for the cy8c9520a is done
cy8c9520a 1-0020: the setup for the cy8c9520a gpio controller done
cy8c9520a 1-0020: the cy8c9520a_irq_setup function is entered
cy8c9520a 1-0020: the interrupt state registers are cleared
cy8c9520a 1-0020: the interrupt mask port registers are set
cy8c9520a 1-0020: the interrupt setup is done
cy8c9520a 1-0020: the interrupt setup for the cy8c9520a is done
cy8c9520a 1-0020: the setup for the cy8c9520a pwm_chip controller is done
cy8c9520a 1-0020: cygpio_pinconf_set function is called
cy8c9520a 1-0020: The pin 0 drive mode is PIN_CONFIG_BIAS_PULL_UP
cy8c9520a 1-0020: cygpio_pinconf_set function is called
cy8c9520a 1-0020: The pin 1 drive mode is PIN_CONFIG_BIAS_PULL_UP
cy8c9520a 1-0020: cygpio_pinconf_set function is called
cy8c9520a 1-0020: The pin 2 drive mode is PIN_CONFIG_BIAS_PULL_DOWN
cy8c9520a 1-0020: cygpio_pinconf_set function is called
cy8c9520a 1-0020: The pin 3 drive mode is PIN_CONFIG_DRIVE_STRENGTH
cy8c9520a 1-0020: the setup for the cy8c9520a pinctl descriptor is done
```

Handle GPIO INT in line 0 of P0 using the gpio interrupt driver.

Load the gpio interrupt module:

```
root@raspberrypi:/home/pi# insmod int_rpi3_gpio.ko
int_gpio_expand int_gpio: my_probe() function is called.
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
int_gpio_expand int_gpio: IRQ_using_platform_get_irq: 61
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
cy8c9520a 1-0020: gpio 0 is unmasked
cy8c9520a 1-0020: the REG_INTR_MASK value is 254
int_gpio_expand int_gpio: mydev: got minor 59
int_gpio_expand int_gpio: my_probe() function is exited.
```

Connect pin 0 of P0 to GND, then disconnect it from GND. Two interrupts are fired:

```
root@raspberrypi:/home/pi#
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
```

Each probed PWM controller will be exported as pwmchipN, where N is the base of the PWM controller:

```
root@raspberrypi:/sys/class/pwm# ls
```

```
pwmchip0
```

```
root@raspberrypi:/sys/class/pwm# cd pwmchip0/
```

npwm is the number of PWM channels this controller supports (read-only):

```
root@raspberrypi:/sys/class/pwm/pwmchip0# ls
device export npwm power subsystem uevent unexport
```

Exports a PWM channel (**pwm1**) with **sysfs** (write-only). The PWM channels are numbered using a per-controller index from 0 to **npwm-1**:

```
root@raspberrypi:/sys/class/pwm/pwmchip0# echo 1 > export
cy8c9520a 1-0020: cy8c9520a_pwm_request is called
```

You can see that the **pwm1** channel has been created. This channel corresponds to the pin 3 of our device:

```
root@raspberrypi:/sys/class/pwm/pwmchip0# ls
device export npwm power pwm1 subsystem uevent unexport
```

Set the total period of the PWM signal (read/write). Value is in nanoseconds:

```
root@raspberrypi:/sys/class/pwm/pwmchip0# echo 100000 > pwm1/period
cy8c9520a 1-0020: cy8c9520a_pwm_config is called
```

Set the active time of the PWM signal (read/write). Value is in nanoseconds:

```
root@raspberrypi:/sys/class/pwm/pwmchip0# echo 50000 > pwm1/duty_cycle
cy8c9520a 1-0020: cy8c9520a_pwm_config is called
```

Enable the PWM signal (0 = disabled and 1 = enabled):

```
root@raspberrypi:/sys/class/pwm/pwmchip0# echo 1 > pwm1/enable
cy8c9520a 1-0020: cy8c9520a_pwm_enable is called
cy8c9520a 1-0020: cy8c9520a_gpio_direction output is called
cy8c9520a 1-0020: cy8c9520a_gpio_set_value func with 1 value is called
```

Connect pin 0 of P0 to pin 3 of P0. You will see how interrupts are being fired in each level change of the PWM signal:

```
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
the interrupt ISR has been entered
int_gpio_expand int_gpio: interrupt received. key: P0_line0_INT
the interrupt ISR has been entered
```

```
[...]
```

Remove the **gpio int** module:

```
root@raspberrypi:/home/pi# rmmod int_rpi3_gpio.ko
int_gpio_expand int_gpio: my_remove() function is called.
```

```

int_gpio_expand int_gpio: my_remove() function is exited.
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_mask is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
cy8c9520a 1-0020: gpio 0 is unmasked
cy8c9520a 1-0020: the REG_INTR_MASK value is 255
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called

```

Remove the CY8C9520A_pwm_pinctrl module:

```

root@raspberrypi:/home/pi# rmmod CY8C9520A_pwm_pinctrl.ko
cy8c9520a 1-0020: cy8c9520a_remove() function is called

```

LAB 7.6: CY8C9520A Device Tree overlay

In this lab, you will see how to introduce new hardware support on Raspberry Pi by using the Raspberry Pi specific Device Tree overlay mechanism. In previous labs, you described the CY8C9520A device by writing its Device Tree properties in the `bcm2710-rpi-3-b.dts` file, which describes the hardware on the Raspberry Pi 3 Model B board. The Device Tree overlay mechanism allows to integrate new hardware, keeping the original Device Tree source files. The Device Tree overlays allow you to override specific parts of a Device Tree and insert dynamically Device Tree fragments to a live tree and effect change.

Open the `bcm2710-rpi-3-b.dts` file (located in `/arch/arm/boot/dts/` in the kernel source tree), and find the `i2c1` nodes. You will see the following nodes:

```

&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
};

i2c1_pins: i2c1 {
    brcm,pins = <2 3>;
    brcm,function = <4>;
};

```

The first node adds several properties to the `i2c1` controller master node. In the `i2c1` node, you can see the `pinctrl-0` property that points to the `i2c1_pins` pin configuration node (second node), which configures the pins of the `i2c1` controller in I2C mode.

The `i2c1` controller node is described in the `bcm283x.dtsi` file (included in `/arch/arm/boot/dts/` in the kernel source tree):

```

i2c1: i2c@7e804000 {
    compatible = "brcm,bcm2835-i2c";
    reg = <0x7e804000 0x1000>;
    interrupts = <2 21>;
    clocks = <&clocks BCM2835_CLOCK_VPU>;
};

```

```

#address-cells = <1>;
#size-cells = <0>;
status = "disabled";
};

```

In the LAB 7.5, you added the following sub-nodes and properties (in bold) to the i2c1 and gpio nodes (included in the bcm2710-rpi-3-b.dts file):

```

&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
    status = "okay";

    cy8c9520a: cy8c9520a@20 {
        compatible = "cy8c9520a";
        reg = <0x20>;
        interrupt-controller;
        #interrupt-cells = <2>;
        gpio-controller;
        #gpio-cells = <2>;

        interrupts = <23 1>;
        interrupt-parent = <&gpio>;

        #pwm-cells = <2>;
        pwm0 = <20>; // pwm not supported
        pwm1 = <3>;
        pwm2 = <20>; // pwm not supported
        pwm3 = <2>;

        pinctrl-names = "default";
        pinctrl-0 = <&accel_int_pin &cy8c9520apullups &cy8c9520apulldowns
&cy8c9520adrivestrength>;

        cy8c9520apullups: pinmux1 {
            pins = "gpio0", "gpio1";
            bias-pull-up;
        };

        cy8c9520apulldowns: pinmux2 {
            pins = "gpio2";
            bias-pull-down;
        };

        /* pwm channel */
        cy8c9520adrivestrength: pinmux3 {
            pins = "gpio3";
            drive-strength;
        };
    };
};

```



```

&gpio {
    spi0_pins: spi0_pins {
        brcm,pins = <9 10 11>;
        brcm,function = <4>; /* alt0 */
    };

    spi0_cs_pins: spi0_cs_pins {
        brcm,pins = <8 7>;
        brcm,function = <1>; /* output */
    };

    i2c0_pins: i2c0 {
        brcm,pins = <0 1>;
        brcm,function = <4>;
    };

    i2c1_pins: i2c1 {
        brcm,pins = <2 3>;
        brcm,function = <4>;
    };

    [...]

    accel_int_pin: accel_int_pin {
        brcm,pins = <23>;
        brcm,function = <0>; /* Input */
        brcm,pull = <0>; /* none */
    };
};

```

Now, you will remove or comment the previous code in bold to test the Device Tree overlay developed in this LAB 7.6. You will also comment the `int_gpio` node, as shown in the following code snippet, to avoid errors during the compilation of the Device Tree. In this lab, you will only request CY8C9520A interrupts from user space using the `gpio_int` application.

```

/* int_gpio {
    compatible = "arrow,int_gpio_expand";
    pinctrl-names = "default";
    interrupt-parent = <&cy8c9520a>;
    interrupts = <0 IRQ_TYPE_EDGE_BOTH>;
}; */

```

As it has been commented previously, the purpose of the Device Tree overlay is to keep our original Device Tree intact and dynamically add the necessary fragments that describe our new hardware. A DT overlay comprises a number of fragments, each of which targets one node and its subnodes. You will add the previous code in bold by using two fragments. Each fragment will consist of two parts: a `target-path` property, with the absolute path to the node that the fragment is going to modify, or a `target` property with the relative path to the node alias (prefixed with an

ampersand symbol) that the fragment is going to modify, and the `__overlay__` itself, the body of which is added to the target node. In our Device Tree overlay:

- `fragment@0` is adding the `accel_int_pin` node to the `gpio` node.
- `fragment@1` is adding the `cy8c9520a` node to the `i2c1` node, and it is also modifying some properties (like the `status` property) of the `i2c1` node itself.

You will create a `cy8c9520a-overlay.dts` file and add the code below, then you will include the file in the `arch/arm/boot/dts/overlays/` folder in the kernel source tree:

```
/ {
    compatible = "brcm,bcm2835";

    fragment@0 {
        target = <&gpio>;
        __overlay__ {
            accel_int_pin: accel_int_pin {
                brcm,pins = <23>;
                brcm,function = <0>;    /* Input */
                brcm,pull = <0>;        /* none */
            };
        };
    };

    fragment@1 {
        target = <&i2c1>;
        __overlay__ {
            #address-cells = <1>;
            #size-cells = <0>;
            status = "okay";

            cy8c9520a: cy8c9520a@20 {
                compatible = "cy8c9520a";
                reg = <0x20>;
                interrupt-controller;
                #interrupt-cells = <2>;
                gpio-controller;
                #gpio-cells = <2>;

                interrupts = <23 1>;
                interrupt-parent = <&gpio>;

                #pwm-cells = <2>;
                pwm0 = <20>; // pwm not supported
                pwm1 = <3>;
                pwm2 = <20>; // pwm not supported
                pwm3 = <2>;

                pinctrl-names = "default";
                pinctrl-0 = <&accel_int_pin &cy8c9520apullups &cy8c9520apulldowns
&cy8c9520adrivestrength>;
            };
        };
    };
};
```

```

        cy8c9520apullups: pinmux1 {
            pins = "gpio0", "gpio1";
            bias-pull-up;
        };

        cy8c9520apulldowns: pinmux2 {
            pins = "gpio2";
            bias-pull-down;
        };

        /* pwm channel */
        cy8c9520adrivestrength: pinmux3 {
            pins = "gpio3";
            drive-strength;
        };
    };
};
};

```

The overlay will get compiled into a .dtbo file. To be compiled, the overlay needs to be referenced in the arch/arm/boot/dts/overlays/Makefile file in the kernel source tree:

```

[...]
upstream.dtbo \
upstream-pi4.dtbo \
vc4-fkms-v3d.dtbo \
vc4-kms-kippah-7inch.dtbo \
vc4-kms-v3d.dtbo \
vc4-kms-v3d-pi4.dtbo \
vga666.dtbo \
w1-gpio.dtbo \
w1-gpio-pullup.dtbo \
w5500.dtbo \
wittypi.dtbo \
cy8c9520a.dtbo

```

With this overlay in place, you need to enable it in the config.txt file, as well as the I2C1 overlay with a correct pin-muxing configuration. You can modify the config.txt file directly in the Raspberry Pi using Nano or Vim editors:

```

root@raspberrypi:/home/pi# cd /boot
root@raspberrypi:/boot# nano config.txt

dtparam=i2c_arm=on
#dtparam=i2s=on
dtparam=spi=on
dtoverlay=spi0-cs
enable_uart=1
kernel=kernel7.img
dtoverlay=cy8c9520a
dtoverlay=i2c1,pins_2_3

```

You can see below the I2C1 overlay (i2c1-overlay.dts), which is included in the arch/arm/boot/dts/overlays/ folder:

```
/dts-v1/;
/plugin/;

/{
    compatible = "brcm,bcm2835";

    fragment@0 {
        target = <&i2c1>;
        __overlay__ {
            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&i2c1_pins>;
        };
    };

    fragment@1 {
        target = <&i2c1_pins>;
        pins1: __overlay__ {
            brcm,pins = <2 3>;
            brcm,function = <4>; /* alt 0 */
        };
    };

    fragment@2 {
        target = <&i2c1_pins>;
        pins2: __dormant__ {
            brcm,pins = <44 45>;
            brcm,function = <6>; /* alt 2 */
        };
    };

    fragment@3 {
        target = <&i2c1>;
        __dormant__ {
            compatible = "brcm,bcm2708-i2c";
        };
    };

    __overrides__ {
        pins_2_3 = <0>, "1!2";
        pins_44_45 = <0>, "!1=2";
        combine = <0>, "!3";
    };
};
```

To avoid the need for lots of Device Tree overlays, and to reduce the need for users of peripherals to modify DT files, the Raspberry Pi loader supports a new feature - Device Tree parameters. These parameters are defined in the DT by adding an `__overrides__` node to the root. You can read about the different types of parameters in the following link of the Raspberry Pi documentation:

<https://www.raspberrypi.org/documentation/configuration/device-tree.md>

The I2C1 overlay will use overlay/fragment parameters. The DT parameter mechanism has a number of limitations, including the inability to change the name of a node and to write arbitrary values to arbitrary properties when a parameter is used. One way to overcome some of these limitations is to conditionally include or exclude certain fragments. A fragment can be excluded from the final merge process (disabled) by renaming the `__overlay__` node to `__dormant__`. The parameter declaration syntax has been extended to allow the otherwise illegal zero target phandle to indicate that the following string contains operations at fragment or overlay scope. So far, four operations have been implemented:

```
+<n>    // Enable fragment <n>
-<n>    // Disable fragment <n>
=<n>    // Enable fragment <n> if the assigned parameter value is true, otherwise disable it
!<n>    // Enable fragment <n> if the assigned parameter value is false, otherwise disable it
```

See below in bold the meaning of the properties included in the `__overlay__` node of the I2C1 overlay:

```
__overrides__ {
    pins_2_3    = <0>,"=1!2"; // the pins_2_3 parameter enables fragment 1 and disables
fragment 2 if value is true
    pins_44_45 = <0>,"!1=2"; // the pins_44_45 parameter enables fragment 2 and
disables fragment 1 if value is true
    combine = <0>,"!3"; // the combine parameter enables fragment 3 if the assigned
parameter value is false, otherwise disable it
};
```

Compile the modified Device Tree and Device Tree overlay, and copy them to the Raspberry Pi device:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
~/linux_rpi3/linux$ scp arch/arm/boot/dts/overlays/cy8c9520a.dtbo root@10.0.0.10:/boot/overlays
```

Compile the `CY8C9520A_pwm_pinctrl.c` driver and the `gpio_int.c` application, and send them to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/device_tree_overlays/linux_5.4_CY8C9520A_pwm_pinctrl$ make
~/linux_5.4_rpi3_drivers/device_tree_overlays/linux_5.4_CY8C9520A_pwm_pinctrl$ make deploy
~/linux_5.4_rpi3_drivers/device_tree_overlays/linux_5.4_CY8C9520A_pwm_pinctrl/app$ scp
gpio_int.c root@10.0.0.10:/home
```

Compile the `gpio_int.c` application on the Raspberry Pi:

```
root@raspberrypi:/home/pi# gcc -o gpio_int gpio_int.c
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

LAB 7.6 driver demonstration

Load the CY8C9520A_pwm_pinctrl.ko module:

```
root@raspberrypi:/home/pi# insmod CY8C9520A_pwm_pinctrl.ko
```

Handle GPIO INT in line 0 of P0 using the gpio_int application. Connect pin 0 of P0 to GND, then remove it from GND. Two interrupts are fired:

```
root@raspberrypi:/home/pi# ./gpio_int
cy8c9520a 1-0020: cy8c9520a_gpio_direction input is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_lock is called
cy8c9520a 1-0020: cy8c9520a_irq_set_type is called
cy8c9520a 1-0020: cy8c9520a_irq_unmask is called
cy8c9520a 1-0020: cy8c9520a_irq_bus_sync_unlock is called
the interrupt ISR has been entered
cy8c9520a 1-0020: cy8c9520a_gpio_get function is called
cy8c9520a 1-0020: the in_reg address is 0
cy8c9520a 1-0020: cy8c9520a_gpio_get function with 254 value is returned
irq received.
id: 2, timestamp: 1606046401377764044
the interrupt ISR has been entered
```

Exit the application with ^C:

```
root@raspberrypi:/home/pi#
```

Remove the CY8C9520A_pwm_pinctrl module:

```
root@raspberrypi:/home/pi# rmmod CY8C9520A_pwm_pinctrl.ko
```

Allocating Kernel Memory

Linux is a virtual memory system, meaning that the addresses seen by user programs do not directly correspond to the physical addresses used by the hardware. Kernel and user processes use virtual addresses, and address translation is done in the hardware **MMU** (Memory Management Unit). With virtual memory, programs running on the system can allocate far more memory than is physically available; indeed, even a single process can have a virtual address space larger than the system's physical memory.

The ARM architecture uses **translation tables** stored in memory to translate virtual addresses to physical addresses. The MMU will automatically read the translation tables when necessary, this process is known as a **Table Walk**.

An important function of the MMU is to enable the system to run multiple tasks as independent programs running in their own private virtual memory space, in many cases sharing virtual addresses. They do not need any knowledge of the physical memory map of the system, that is, the addresses that are used by the hardware, or about other programs that might execute at the same time. You can use the same virtual memory address space for each user program. You can also work with a contiguous virtual memory map, even if the physical memory is fragmented. You can write, compile and link applications to run in the virtual memory space. Physical addresses are those used by the actual hardware system.

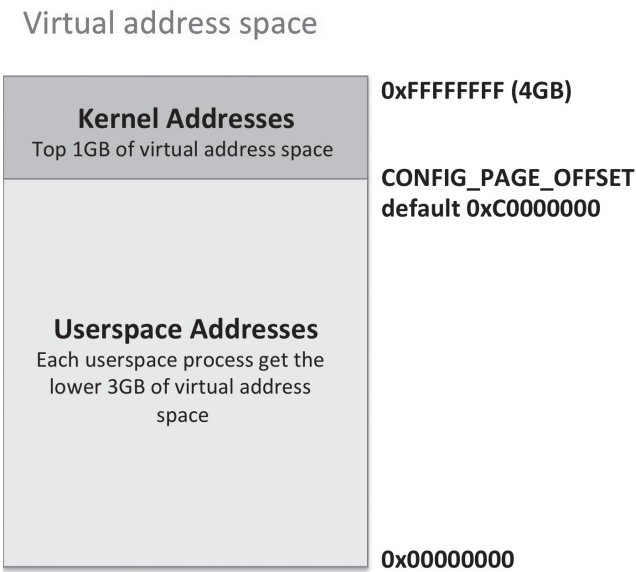
When a process tries to access memory in a page that is not known to the MMU, the MMU generates a page fault exception. The page fault exception handler examines the state of the MMU hardware and the currently running process's memory information and determines whether the fault is a "good" one, or a "bad" one. Good page faults cause the handler to give more memory to the process; bad faults cause the handler to terminate the process.

Linux address types

The following is a list of address types used in Linux:

1. **User virtual addresses** -- These are the regular addresses seen by user space programs. User addresses are either 32 or 64 bits in length, depending on the underlying hardware

architecture, and each process has its own virtual address space. The virtual address space is split; the lower part is used for user space and the upper part is used for the kernel. If you assign 1GB of virtual address space for the kernel on 32-bit processors, the split is at 0xC0000000.



- 2. **Physical addresses** -- These are the addresses used between the processor and the system's memory. Physical addresses are 32-bit or 64-bit quantities; even 32-bit systems can use larger physical addresses in some situations.
- 3. **Bus addresses** -- These are the addresses used between peripheral buses and memory. Often, they are the same as the physical addresses used by the processor, but that is not necessarily the case. Some architectures can provide an I/O memory management unit, IOMMU, that remaps addresses between a bus and main memory. Programming the IOMMU is an extra step that must be performed when setting up DMA operations.
- 4. **Kernel logical addresses** -- These make up the normal address space of the kernel. These are the virtual addresses above CONFIG_PAGE_OFFSET. On most architectures, logical addresses and their associated physical addresses differ only by a constant offset which makes converting between physical and virtual addresses easy. The kmalloc() function returns a pointer variable that points to a kernel logical address space that is mapped to continuous physical pages. The kernel logical memory cannot be swapped out. Kernel

logical addresses can be converted to and from physical addresses by using the macros `__pa(x)` and `__va(x)`.

5. **Kernel virtual addresses** -- These addresses are similar to logical addresses in that they are a mapping from a kernel space address to a physical address. Kernel virtual addresses do not necessarily have the linear, one-to-one mapping to physical addresses that characterize the logical address space, however. All logical addresses are kernel virtual addresses, but many kernel virtual addresses are not logical addresses; for example, the function `vmalloc()` will return a block of virtual memory which is continuous in virtual space, but it may not be continuous in physical space. Memory returned by `ioremap()` will be dynamically placed in the kernel virtual region. Machine specific static mappings are also located here, through `iotable_init()`.

User process virtual to physical memory mapping

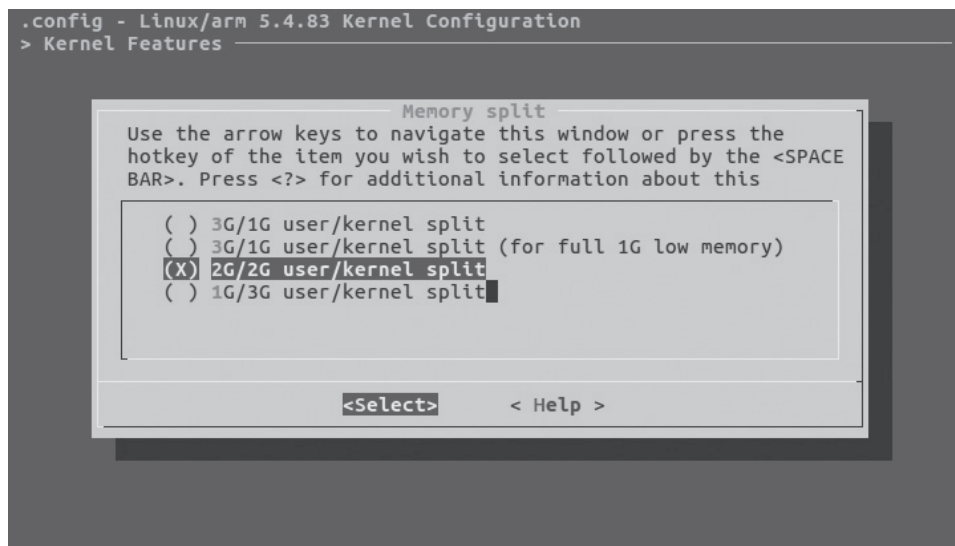
In Linux, kernel space is where kernel executes and provides its services. Kernel space is protected so that user applications can not access it directly, while user space can be directly accessed from code running in kernel mode. Kernel space is located at the top of the virtual address space, while user space is located at the bottom. The kernel creates mappings that prevent access to kernel space from user mode.

Every user space process has its own virtual memory layout, containing four logical areas:

1. **Text segment** -- Program code, stores the binary image of the process (`./bin/app`).
2. **Data segment (data+bss+heap)** -- Various data structures created and initialized at the start of a process or while it is running (e.g., heap). The heap provides runtime memory allocation, like the stack, meant for data that must outlive the function doing the allocation, unlike the stack. In C, the main interface to heap allocation is the `malloc()` function. A data segment stores static initialized variables and BSS segment uninitialized static variables filled with zeros.
3. **Memory mapping segment** -- In this segment, the kernel maps the contents of files directly to memory. This mapping can be done by calling the Linux `mmap()` system call.
4. **Stack segment** -- Starts near the end of the area available to process and grows downwards. Stores local variables and function parameters in most programming languages. Calling a method or function pushes a new stack frame onto the stack. The stack frame is destroyed when the function returns. Each thread in a process gets its own stack.

Kernel virtual to physical memory mapping

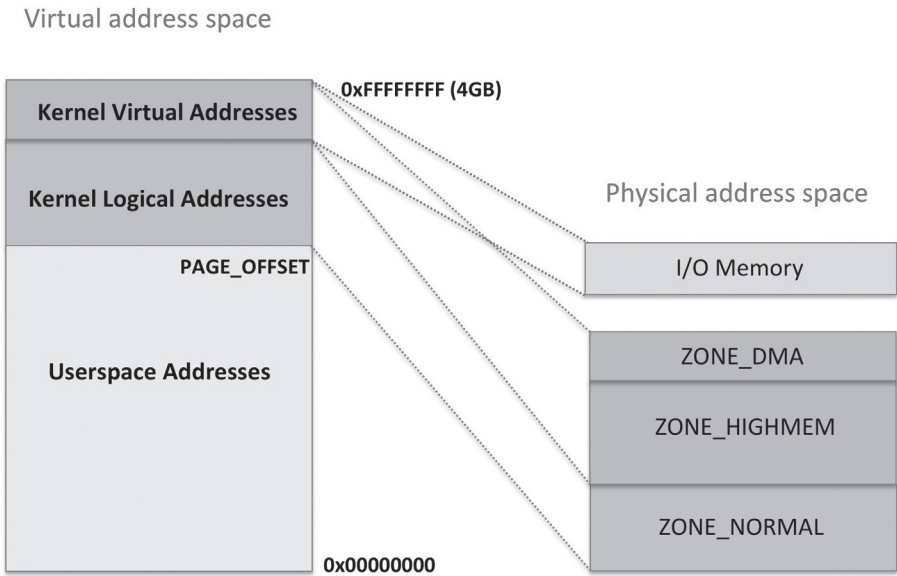
The kernel virtual address space starts from 0xc0000000. You can go to the kernel config settings to allow the kernel to access more physical memory:



You can split the kernel physical memory into four zones:

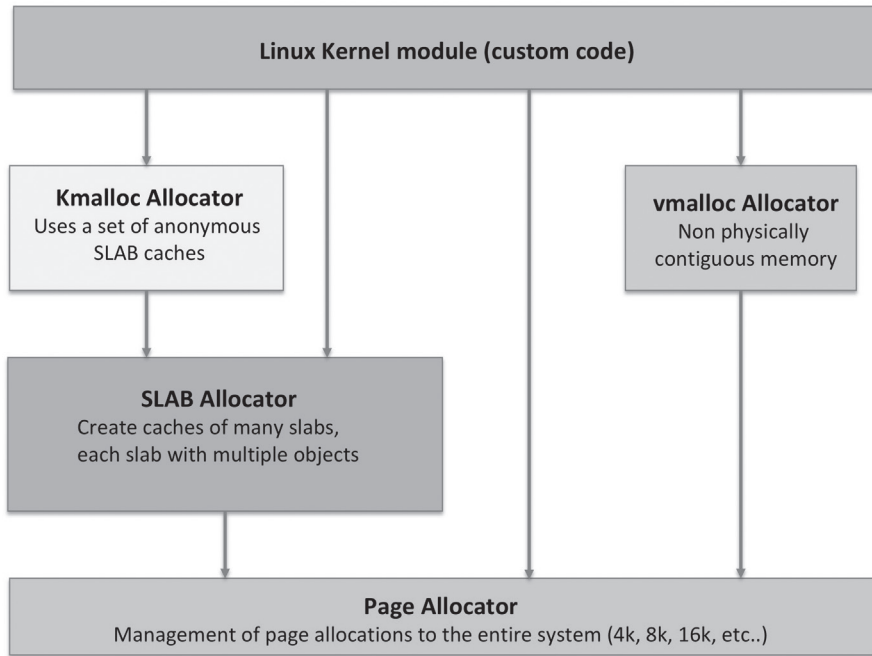
1. **ZONE_DMA** -- Mapped to the kernel virtual address space (HIGHMEM). The mapped virtual DMA memory region is returned by the `dma_alloc_xxx` functions.
2. **ZONE_NORMAL** -- Mapped to the kernel logical address space (LOWMEM). Used by the kernel for internal data structures as well as other system and user space allocations. `kmalloct()` is a memory-allocation function that returns contiguous memory from ZONE_NORMAL.
3. **ZONE_HIGHMEM** -- Mapped to the kernel virtual address space (HIGHMEM). Used exclusively for system allocations (file system buffers, user space allocations, etc.). Mapped to kernel virtual addresses returned by the `vmalloc` function.
4. **Memory-Mapped I/O** -- Mapped to kernel virtual address space (HIGHMEM). Memory returned by `ioremap()` will be dynamically placed in this kernel virtual region.

In the next figure, you can see the kernel memory mapping layout:



Kernel memory allocators

The Linux kernel provides a few memory allocation methods. The main one is the **Page Allocator** that works on pages. The **SLAB allocator** is built upon page allocator, getting memory from it and handling it using smaller entities. Kernel memory allocators allocate physical pages, and kernel allocated memory cannot be swapped out, so no fault handling is required. Most kernel memory allocation functions also return a pointer to a kernel virtual address to be used within kernel space.



Page allocator

The Page allocator is responsible for the management of page allocations to the entire system. This code manages lists of physically contiguous pages and maps them into the MMU page tables, so as to provide other kernel subsystems with valid physical address ranges when the kernel requests them (physical to virtual address mapping is handled by a higher layer of the VM). The principal algorithm used for page allocations is the Binary Buddy Allocator. This allocator is explained in great detail in the kernel documentation at the following location:

<https://www.kernel.org/doc/gorman/html/understand/understand009.html>

Page allocator API

To allocate pages, these are some of the available functions:

```

unsigned long get_zeroed_page(int flags); /* Returns the virtual address of a free page,
initialized to zero */

unsigned long __get_free_page(int flags); /* Same, but doesn't initialize the contents */

unsigned long __get_free_pages(int flags, unsigned int order); /* Returns the starting
virtual address of an area of several contiguous pages in physical RAM, with the order being
Log2(number_of_pages. Can be computed from the size with the get_order() function */

```

These are the most common flags:

- **GFP_KERNEL**: Standard kernel memory allocation. The allocation may block in order to find enough available memory. Fine for most needs, except in an interrupt handler context.
- **GFP_ATOMIC**: RAM allocated from code which is not allowed to block (interrupt handlers or critical sections). Never blocks, allows to access emergency pools, but can fail if no free memory is readily available.
- **GFP_DMA**: Allocates memory in an area of the physical memory that is used for DMA transfers.

SLAB allocator

The SLAB Allocator allows creation of "caches", one cache for each object type (for example, `inode_cache`, `dentry_cache`, `buffer_head`, `vm_area_struct`). Each cache consists of many "slabs" (usually one page long and always contiguous), and each slab contains multiple initialized objects.

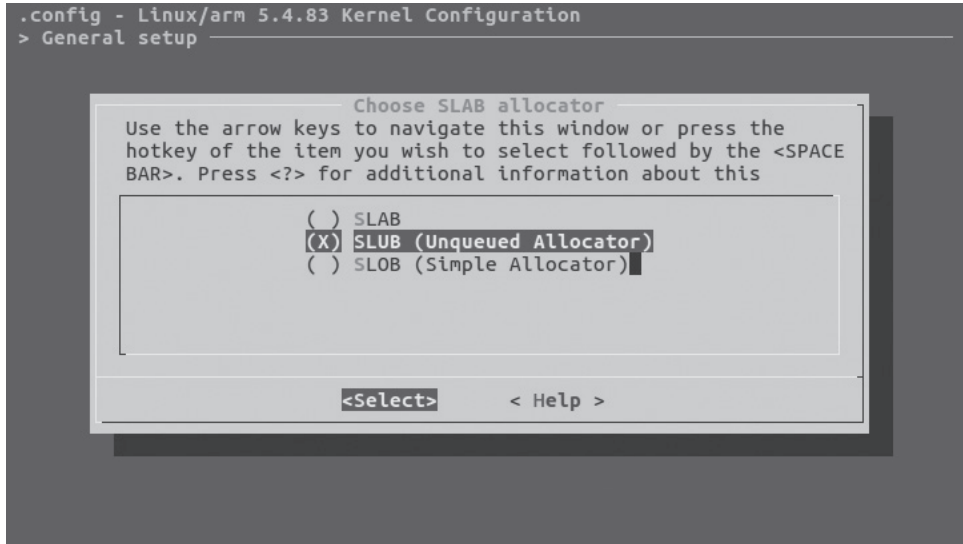
The primary intention of the slab allocation technique was to efficiently manage the allocation of kernel objects and prevent memory fragmentation caused by memory allocation and deallocation. The kernel objects are the allocated and initialized objects of the same type that are usually represented in the form of a structure in C. These objects are only used by the kernel core, modules and drivers that run in kernel space. The object size can be smaller or greater than the page size. The SLAB allocator takes care of increasing or reducing the size of the cache as needed, depending on the number of allocated objects, using the page allocator to allocate and free pages.

The SLAB allocator consists of a variable number of caches that are linked together in a doubly linked circular list called a "cache chain". In order to reduce fragmentation, the slabs are sorted into three groups:

- Full slabs with zero free objects.
- Partial slabs.
- Empty slabs with no allocated objects.

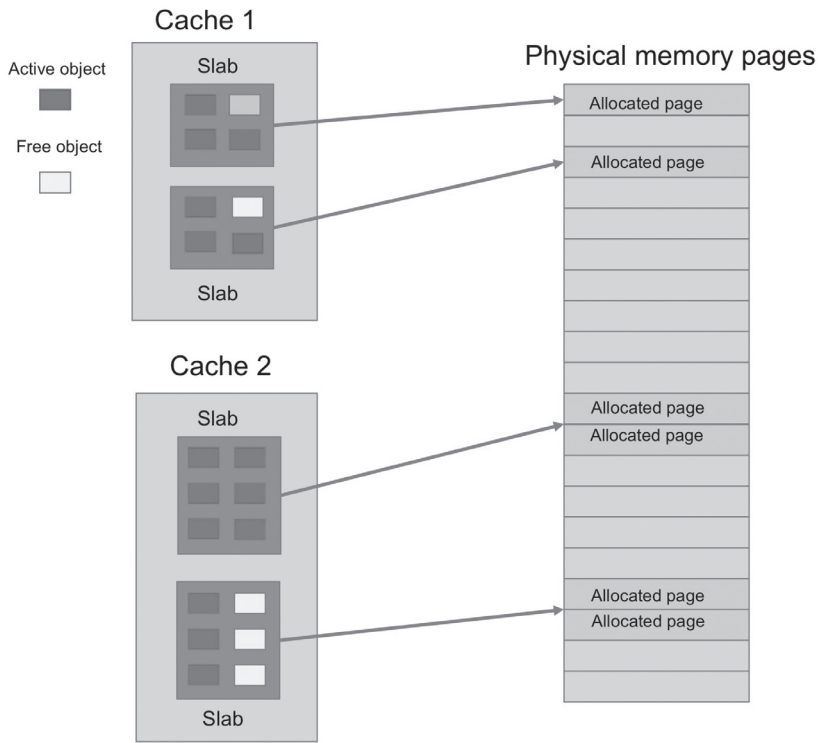
If partial slabs exist, then new allocations come from these slabs, otherwise from empty slabs or new slabs are allocated.

In the Linux kernel there are three different implementations of the slab allocation technique, namely SLAB, SLUB and SLOB:



- **CONFIG_SLAB:** Legacy.
- **CONFIG_SLOB:** Simple allocator, saves about 0.5MB of memory, but does not scale well. It is used for very small systems with limited memory (option activates after selecting CONFIG_EMBEDDED).
- **CONFIG_SLUB:** Default since 2.6.23. Simpler than SLAB, scales better.

In the following figure, you can see the general memory layout of the SLAB allocator:



To understand Linux kernel SLAB allocators, the following terms are defined, which appear frequently in the SLAB allocator source code:

1. **Cache** is a group of the kernel objects of the same type. Cache is identified by a name that is usually the same as the C structure name. The kernel uses a doubly-linked list to link the created caches.
2. **Slab** is the contiguous block of memory stored in one or more physical page(s) of the main memory. Each cache has a number of slabs that store the actual kernel objects of the same type.
3. **Kernel object** is the allocated and initialized instance of a C structure. Each slab may contain some objects (depending on the size of the slab and each object). A kernel object in the slab can be either active (object is being used by the kernel) or free (the object is in the memory pool and ready to be used upon request).

SLAB allocator API

The Linux kernel SLAB allocation sub-system provides a general interface for creating and destroying a memory cache regardless of the type of SLAB allocator:

1. The `kmem_cache_create()` function creates a new memory cache:

```
struct kmem_cache *kmem_cache_create(const char *name, size_t size,
                                     size_t align, unsigned long flags,
                                     void (*ctor)(void*));
```

Taking the following parameters:

- **name:** A string which is used in `/proc/slabinfo` to identify this cache.
 - **size:** The size of objects to be created in this cache.
 - **align:** Additional space added to each object (for some additional data).
 - **flags:** SLAB flags.
 - **constructor:** Used to initialize objects.
2. The `kmem_cache_destroy()` function allows destruction of a memory cache by providing the `kmem_cache` object of the desired cache:

```
void kmem_cache_destroy(struct kmem_cache *cp);
```

SLOB, SLAB and SLUB allocators provide two functions for allocating (taking from cache) and freeing (putting back into the cache) a kernel object. Defined as a function in `mm/slab.c`, these are `mm/slob.c` or `mm/slab.c` depending on the chosen slab technique.

1. The `kmem_cache_alloc()` function allocates an object of a specified type from a cache (a cache for that specified object must be created before allocation):

```
void *kmem_cache_alloc(struct kmem_cache *s, gfp_t gfpflags);
```

2. The `kmem_cache_free()` function frees an object and put it back in the cache:

```
void kmem_cache_free(struct kmem_cache *s, void *x);
```

Assume that a Linux kernel module needs to allocate and release an object of a particular type often. The module makes a request to the SLAB allocator by calling the `kmem_cache_create()` function, which creates a cache of that structure type so that it can satisfy subsequent memory allocations (and releases). Based on the size of the structure, the SLAB allocator calculates the number of memory pages required for storing each slab cache (power of 2) and the number of objects that can be stored on each slab. Then, it returns a pointer of type `kmem_cache` as a reference to the created cache.

At the time of creating a new cache, the SLAB allocator generates a number of slabs and populates them with the allocated and initialized objects. When the creation of a new object of the same type is needed, it

makes a request to the SLAB allocator by calling the `kmem_cache_alloc()` function with a pointer (of type `kmem_cache`) to the cache. If the cache has a free object, it immediately returns it. However, if all objects within the cache slabs are already in use (active), the SLAB allocator increases the cache by making a request to the Page allocator (calling the `alloc_pages()` function) to get free pages. After receiving free pages from the Page allocator, the SLAB allocator creates one or more slabs (in the free physical pages) and populates them with the new allocated and initialized objects. On the other hand, at the time of releasing the active object, it is called the `kmem_cache_free()` function with the cache and object pointers as the parameters. The SLAB allocator marks the object as free and keeps the object in the cache for the subsequent requests.

Kmalloc allocator

This is the allocator for the driver code. For large sizes, it relies on the page allocator. For smaller sizes, it relies on generic SLAB caches, named `Kmalloc-xxx` in `/proc/slabinfo`. The allocated area is guaranteed to be physically contiguous and uses the same flags as the page allocator (`GFP_KERNEL`, `GFP_ATOMIC`, `GFP_DMA`, etc.). Maximum sizes for ARM are 4 MB per allocation and 128MB for total allocations. The `kmalloc` allocator should be used as the primary allocator unless there is a strong reason to use another one. See the `kmalloc` allocator API below:

1. Allocate memory by using `kmalloc()` or `kzalloc()` functions:

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

static inline void *kmalloc(size_t size, int flags)
void *kzalloc(size_t size, gfp_t flags) /* Allocates a zero-initialized buffer */
```

These functions allocate `size` bytes and return a pointer to the virtual memory area. The `size` parameter is the number of bytes to allocate, and the `flags` parameter uses the same variants as the page allocator.

You will use the `kfree()` function to free a block of memory allocated with `kmalloc()`:

```
void kfree(const void *objp);
```

2. Conforming to unified device model, memory allocations can be attached to the device. The `devm_kmalloc()` function is a resource-managed `kmalloc()`:

```
/* Automatically free the allocated buffers when the corresponding
device or module is unprobed */
void *devm_kmalloc(struct device *dev, size_t size, int flags);

/* Allocates a zero-initialized buffer */
void *devm_kzalloc(struct device *dev, size_t size, int flags);

/* Useful to immediately free an allocated buffer */
void *devm_kfree(struct device *dev, void *p);
```

LAB 8.1: "linked list memory allocation" module

In this lab, you will allocate in the kernel memory a circular single linked list composed of several nodes. Each node will be composed of two variables:

1. A buffer pointer that points to a memory buffer allocated with `devm_kmalloc()` using a "for" loop.
2. A next pointer that points to the next node of the linked list.

The linked list will be managed through the items of a structure named `liste`. The driver's `write()` callback function will get the characters written to the user space console. These characters fill each node buffer of the linked list starting from the first member. It will be moved to the next node when the node is filled with the selected buffer size (variable `BlockSize`). The driver will write again to the first node buffer of the linked list when the last node buffer has been filled.

You can read all the values written to the nodes via the `read()` callback function. The reading goes from the first written node buffer to the last written node buffer of the linked list. After exiting the `read()` function, all the `liste` pointers point to the first node of the linked list, and the `cur_read_offset` and `cur_write_offset` variables are set to zero to start writing again from the first node.

The main code sections of the driver will now be described:

1. Create each node of the linked list:

```
typedef struct dnode
{
    char *buffer;
    struct dnode *next;
} data_node;
```

2. Create a `liste` structure to manage the nodes of the linked list:

```
typedef struct lnode
{
    data_node *head;
    data_node *cur_write_node;
    data_node *cur_read_node;
    int cur_read_offset;
    int cur_write_offset;
} liste;
```

3. Allocate the first node of the linked list in the `createlist()` function (`createlist()` is called inside the `probe()` function) using the `devm_kmalloc()` function (all the nodes will be freed automatically when the module is unprobed):

```
/* Allocate the first node */
newNode = devm_kmalloc(&device->dev, sizeof (data_node), GFP_KERNEL);

/* Allocate first node memory buffer */
```

```

newNode->buffer = devm_kmalloc(&device->dev, BlockSize*sizeof(char), GFP_KERNEL);
newNode->next = NULL;
newListe.head = newNode;
headNode = newNode;
previousNode = newNode;

```

4. In the createlist() function, allocate the rest of the linked list nodes up to BlockNumber through a for loop. After the for loop, link the last linked list node with the first one:

```

for (i = 1; i < BlockNumber; i++)
{
    newNode = (data_node *)devm_kmalloc(&device->dev, sizeof (data_node), GFP_KERNEL);
    newNode->buffer = (char *)devm_kmalloc(&device->dev,
                                           BlockSize*sizeof(char),
                                           GFP_KERNEL);

    newNode->next = NULL;
    previousNode->next = newNode;
    previousNode = newNode;
}

newNode->next = headNode;
newListe.cur_read_node = headNode;
newListe.cur_write_node = headNode;
newListe.cur_read_offset = 0;
newListe.cur_write_offset = 0;

```

5. Open the bcm2710-rpi-3-b.dts DT file, and add the linked_memory node in the soc node:

```

&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };
    [...]
    linked_memory {
        compatible = "arrow,memory";
    };
}

```

6. Create a new linkedlist_rpi3_platform.c file in the linux_5.4_rpi3_drivers folder, and add linkedlist_rpi3_platform.o to your Makefile obj-m variable, then build and deploy the module to the Raspberry Pi:

```

~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy

```

7. Build the modified Device Tree, and load it to the target processor:

```

~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs

```

```
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

8. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 8-1: linkedlist_rpi3_platform.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/uaccess.h>
#include <linux/miscdevice.h>
#include <linux/delay.h>

static int BlockNumber = 10;
static int BlockSize = 5;
static int size_to_read = 0;
static int node_count= 1;
static int cnt = 0;

typedef struct dnode
{
    char *buffer;
    struct dnode *next;
} data_node;

typedef struct lnode
{
    data_node *head;
    data_node *cur_write_node;
    data_node *cur_read_node;
    int cur_read_offset;
    int cur_write_offset;
} liste;

static liste newList;

static int createlist (struct platform_device *pdev)
{
    data_node *newNode, *previousNode, *headNode;
    int i;

    /* New node creation */
    newNode = devm_kmalloc(&pdev->dev, sizeof(data_node), GFP_KERNEL);
    if (newNode)
        newNode->buffer = devm_kmalloc(&pdev->dev, BlockSize*sizeof(char), GFP_KERNEL);
    if (!newNode || !newNode->buffer)
        return -ENOMEM;

    newNode->next = NULL;
```

```

newListe.head = newNode;
headNode = newNode;
previousNode = newNode;

for (i = 1; i < BlockNumber; i++)
{
    newNode = devm_kmalloc(&pdev->dev, sizeof(data_node), GFP_KERNEL);
    if (newNode)
        newNode->buffer = devm_kmalloc(&pdev->dev, BlockSize*sizeof(char), GFP_KERNEL);
    if (!newNode || !newNode->buffer)
        return -ENOMEM;
    newNode->next = NULL;
    previousNode->next = newNode;
    previousNode = newNode;
}

newNode->next = headNode;

newListe.cur_read_node = headNode;
newListe.cur_write_node = headNode;
newListe.cur_read_offset = 0;
newListe.cur_write_offset = 0;

return 0;
}

static ssize_t my_dev_write(struct file *file, const char __user *buf,
                           size_t size, loff_t *offset)
{
    int size_to_copy;
    pr_info("my_dev_write() is called.\n");
    pr_info("node_number_%d\n", node_count);

    if ((*offset) == 0 || (node_count == 1))
    {
        size_to_read += size;
    }

    if (size < BlockSize - newListe.cur_write_offset)
        size_to_copy = size;
    else
        size_to_copy = BlockSize - newListe.cur_write_offset;

    if(copy_from_user(newListe.cur_write_node->buffer + newListe.cur_write_offset, buf,
                      size_to_copy))
    {
        return -EFAULT;
    }

    *(offset) += size_to_copy;
    newListe.cur_write_offset += size_to_copy;

    if (newListe.cur_write_offset == BlockSize)

```

```

{
    newListe.cur_write_node = newListe.cur_write_node->next;
    newListe.cur_write_offset = 0;
    node_count = node_count+1;
    if (node_count > BlockNumber)
    {
        newListe.cur_read_node = newListe.cur_write_node;
        newListe.cur_read_offset = 0;
        node_count = 1;
        cnt = 0;
        size_to_read = 0;
    }
}
return size_to_copy;
}

static ssize_t my_dev_read(struct file *file, char __user *buf, size_t count, loff_t *offset)
{
    int size_to_copy;
    int read_value;

    read_value = (size_to_read - (BlockSize * cnt));

    if ((*offset) < size_to_read)
    {
        if (read_value < BlockSize - newListe.cur_read_offset)
            size_to_copy = read_value;
        else
            size_to_copy = BlockSize - newListe.cur_read_offset;
        if(copy_to_user(buf, newListe.cur_read_node->buffer + newListe.cur_read_offset,
            size_to_copy))
        {
            return -EFAULT;
        }
        newListe.cur_read_offset += size_to_copy;
        (*offset)+=size_to_copy;

        if (newListe.cur_read_offset == BlockSize)
        {
            cnt = cnt+1;
            newListe.cur_read_node = newListe.cur_read_node->next;
            newListe.cur_read_offset = 0;
        }
        return size_to_copy;
    }
    else
    {
        msleep(250);
        newListe.cur_read_node = newListe.head;
        newListe.cur_write_node = newListe.head;
        newListe.cur_read_offset = 0;
        newListe.cur_write_offset = 0;
        node_count = 1;
        cnt = 0;
    }
}

```

```

        size_to_read = 0;
        return 0;
    }
}

static int my_dev_open(struct inode *inode, struct file *file)
{
    pr_info("my_dev_open() is called.\n");
    return 0;
}

static int my_dev_close(struct inode *inode, struct file *file)
{
    pr_info("my_dev_close() is called.\n");
    return 0;
}

static const struct file_operations my_dev_fops = {
    .owner = THIS_MODULE,
    .open = my_dev_open,
    .write = my_dev_write,
    .read = my_dev_read,
    .release = my_dev_close,
};

static struct miscdevice helloworld_miscdevice = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "mydev",
    .fops = &my_dev_fops,
};

static int __init my_probe(struct platform_device *pdev)
{
    int ret_val;
    pr_info("platform_probe enter\n");
    createlist(pdev);
    ret_val = misc_register(&helloworld_miscdevice);
    if (ret_val != 0)
    {
        pr_err("could not register the misc device mydev");
        return ret_val;
    }
    pr_info("mydev: got minor %i\n", helloworld_miscdevice.minor);

    return 0;
}

static int __exit my_remove(struct platform_device *pdev)
{
    misc_deregister(&helloworld_miscdevice);
    pr_info("platform_remove exit\n");
    return 0;
}

```

```

static const struct of_device_id my_of_ids[] = {
    { .compatible = "arrow,memory"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "memory",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};

static int demo_init(void)
{
    int ret_val;
    pr_info("demo_init enter\n");

    ret_val = platform_driver_register(&my_platform_driver);
    if (ret_val != 0)
    {
        pr_err("platform value returned %d\n", ret_val);
        return ret_val;
    }
    pr_info("demo_init exit\n");
    return 0;
}

static void demo_exit(void)
{
    pr_info("demo_exit enter\n");
    platform_driver_unregister(&my_platform_driver);
    pr_info("demo_exit exit\n");
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a platform driver that writes in and read \
    from a linked list of several buffers ");

```


linkedlist_rpi3_platform.ko demonstration

Load the module:

```
root@raspberrypi:/home/pi# insmod linkedlist_rpi3_platform.ko
demo_init enter
platform_probe enter
mydev: got minor 60
demo_init exit
```

Write values to the nodes:

```
root@raspberrypi:/home/pi# echo abcdefg > /dev/mydev
my_dev_open() is called.
my_dev_write() is called.
node_number_1
my_dev_write() is called.
node_number_2
my_dev_close() is called.
```

Read values from nodes. After reading, the first node is pointed:

```
root@raspberrypi:/home/pi# cat /dev/mydev
my_dev_open() is called.
abcdefg
my_dev_close() is called.
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod linkedlist_rpi3_platform.ko
demo_exit enter
platform_remove exit
demo_exit exit
```


DMA in Device Drivers

Direct memory access (DMA) is a system controller available on embedded processors that allows for some of its main peripherals (SPI, I2C, UART, general-purpose timers, DAC and ADC) to transfer their I/O data directly to and from main memory independently of the main processing unit. The DMA is also used to manage direct data transfers between RAM data buffers without CPU intervention. While the DMA transfer is in progress, the CPU can continue executing code. When the DMA transfer is completed, the DMA system controller will signal the CPU with an interrupt.

Typical scenarios of block memory copy where DMA can be useful are network packet routing and video streaming applications. DMA is a particular advantage in situations where the blocks to be transferred are larger, or the transfer is a repetitive operation that would consume a large portion of potentially useful CPU processing time.

Cache coherency

One of the main problems when using DMA in a cached system is the possibility that the contents of the cache are not coherent with respect to the system memory. Let's take as an example a CPU with a cache and external memory that can be accessed by peripherals using DMA. When the CPU tries to access data X located in the main memory, it could happen that the current data X value has been cached by the processor, then 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 main memory before the next time a device (DMA) tries to transfer X, the device will receive a stale value of X. Similarly, if the cached copy of X is not invalidated before a device (DMA) writes a new value to the main memory, then the CPU will operate on a stale value of X. Also, when the cache is flushed, the stale data will be written back to the main memory, overwriting the new data stored by the DMA. The end result is that the data in the main memory is not correct.

Some processors include a mechanism called bus snooping or cache snooping. This system notifies to the cache controller the accesses to DMA memory regions, invalidating (DMA reads) or cleaning (DMA writes) the corresponding cache lines. These systems are called **coherent architectures**,

providing a hardware to take care of cache coherency related problem. Hardware will itself maintain coherency between caches and main memory and ensure that all the subsystem (CPU and DMA) have the same view of the memory.

For **non-coherent architectures**, the device driver should explicitly flush or invalidate the data cache before initiating a transfer or making data buffers available to bus mastering peripherals. This can also complicate the software and will cause more transfers between the cache and the main memory, but it does allow the application to use any arbitrary region of cached memory as a data buffer.

Linux kernel provides two `dma_map_ops` structures for ARM processors, one for non-coherent architectures (`arm_dma_ops`) that doesn't provide additional hardware support for coherency management, so software needs to take care of it, and one for coherent ARM architecture (`arm_coherent_dma_ops`) that provides hardware to take care of cache coherency.

```
struct dma_map_ops arm_dma_ops = {
    .alloc          = arm_dma_alloc,
    .free           = arm_dma_free,
    .mmap           = arm_dma_mmap,
    .get_sgtable    = arm_dma_get_sgtable,
    .map_page       = arm_dma_map_page,
    .unmap_page     = arm_dma_unmap_page,
    .map_sg         = arm_dma_map_sg,
    .unmap_sg       = arm_dma_unmap_sg,
    .sync_single_for_cpu = arm_dma_sync_single_for_cpu,
    .sync_single_for_device = arm_dma_sync_single_for_device,
    .sync_sg_for_cpu   = arm_dma_sync_sg_for_cpu,
    .sync_sg_for_device = arm_dma_sync_sg_for_device,
};
EXPORT_SYMBOL(arm_dma_ops);

struct dma_map_ops arm_coherent_dma_ops = {
    .alloc          = arm_coherent_dma_alloc,
    .free           = arm_coherent_dma_free,
    .mmap           = arm_coherent_dma_mmap,
    .get_sgtable    = arm_dma_get_sgtable,
    .map_page       = arm_coherent_dma_map_page,
    .map_sg         = arm_dma_map_sg,
};
EXPORT_SYMBOL(arm_coherent_dma_ops);
```

Linux DMA Engine API

The Linux DMA Engine API specifies an interface to the actual DMA controller hardware functionality to initialize/clean-up and perform DMA transfers.

The DMA Engine API Guide located at <https://www.kernel.org/doc/html/latest/driver-api/dmaengine/client.html> explains in detail the main steps for the slave DMA usage. These are:

- Allocate a DMA slave channel.
- Set slave and controller specific parameters.
- Get a descriptor for the transaction.
- Submit the transaction.
- Issue pending requests and wait for callback notification.

These are the details of the previous steps extracted from the DMA Engine API Guide:

1. **Allocate a DMA slave channel:** Channel allocation is slightly different in the slave DMA context, client drivers typically need a channel from a particular DMA controller only and even in some cases a specific channel is desired. To request a channel, the `dma_request_chan()` API is used.

```
struct dma_chan *dma_request_chan(struct device *dev, const char *name);
```

Which will find and return the DMA channel associated with the `dev` device. A channel allocated via this interface is exclusive to the caller until `dma_release_channel()` is called.

2. **Set slave and controller specific parameters:** Next step is always to pass some specific information to the DMA driver. Most of the generic information which a slave DMA can use is in the `dma_slave_config` structure. This allows the clients to specify DMA direction, DMA addresses, bus widths, DMA burst lengths, etc. for the peripheral. If some DMA controllers have more parameters to be sent, then they should try to embed `dma_slave_config` in their controller specific structure. That gives flexibility to client to pass more parameters, if required.

```
int dmaengine_slave_config(struct dma_chan *chan, struct dma_slave_config *config);
```

3. **Get a descriptor for the transaction:** For slave usage, the various modes of slave transfers supported by the DMA-engine are:
 - **slave_sg:** DMA a list of scatter gather buffers from/to a peripheral.
 - **dma_cyclic:** Performs a cyclic DMA operation from/to a peripheral till the operation is explicitly stopped.
 - **interleaved_dma:** This is common to slave as well as M2M clients. For slave address of devices' fifo could be already known to the driver. Various types of operations could be expressed by setting appropriate values to the `dma_interleaved_template` members.

A non-NULL return of this transfer API represents a "descriptor" for the given transaction.

```
struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(
    struct dma_chan *chan, struct scatterlist *sgl,
    unsigned int sg_len, enum dma_data_direction direction,
    unsigned long flags);

struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(
    struct dma_chan *chan, dma_addr_t buf_addr, size_t buf_len,
    size_t period_len, enum dma_data_direction direction);

struct dma_async_tx_descriptor *dmaengine_prep_interleaved_dma(
    struct dma_chan *chan, struct dma_interleaved_template *xt,
    unsigned long flags);
```

The peripheral driver is expected to have mapped the scatterlist for the DMA operation prior to calling `dmaengine_prep_slave_sg()` and must keep the scatterlist mapped until the DMA operation has completed. The scatterlist must be mapped using the DMA struct device. If a mapping needs to be synchronized later, `dma_sync_*_for_*`() must be called using the DMA struct device, too. So, normal setup should look like this:

```
nr_sg = dma_map_sg(chan->device->dev, sgl, sg_len);
desc = dmaengine_prep_slave_sg(chan, sgl, nr_sg, direction, flags);
```

Once a descriptor has been obtained, the callback information can be added and the descriptor must then be submitted.

4. **Submit the transaction:** Once the descriptor has been prepared and the callback information added, it must be placed on the DMA engine drivers pending queue.

```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc);
```

This returns a cookie that can be used to check the progress of DMA engine activity via other DMA engine calls. The `dmaengine_submit()` call will not start the DMA operation, it merely adds it to the pending queue.

5. **Issue pending DMA requests and wait for callback notification:** The transactions in the pending queue can be activated by calling the `issue_pending` API. If the channel is idle, then the first transaction in the queue is started and subsequent ones queued up. On completion of each DMA operation, the next in the queue is started and a tasklet triggered. The tasklet will then call the client completion callback routine for notification, if set.

```
void dma_async_issue_pending(struct dma_chan *chan);
```

In the Linux DMA API, there are involved different kind of addresses. As you saw in the Chapter 8, the kernel normally uses virtual addresses. Any address returned by `kmalloc()`, `vmalloc()` and similar interfaces is a virtual address. The virtual memory system translates virtual addresses to CPU physical addresses, which are stored as `phys_addr_t` or `resource_size_t`.

The kernel manages processor resources like peripheral registers as physical addresses. These are the addresses seen in `/proc/iomem`. The physical addresses must be mapped to virtual addresses to be useful to a driver. The `ioremap()` function will map the physical addresses producing virtual ones.

If the device supports DMA, the driver sets up a buffer by using `kmalloc()` or a similar interface, which returns a virtual address (X). The virtual memory system maps X to a physical address (Y) in system RAM. The driver can use virtual address X to access the buffer, but the device itself cannot because DMA doesn't go through the CPU virtual memory system. This is part of the reason for the DMA API: the driver can provide a virtual address X to an interface like `dma_map_single()`, which returns the DMA bus address (Z). The driver then tells the device to perform DMA to Z.

The memory accessed by the DMA should be physically contiguous. Any memory allocated by `kmalloc()` (up to 128 KB) or `__get_free_pages()` (up to 8MB) can be used. What cannot be used is `vmalloc()` memory allocation (it would have to set up DMA on each individual physical page).

The **Contiguous Memory Allocator** (or CMA) was developed to allocate big, physically-contiguous memory blocks, commonly being used by DMA Engines. The CMA is integrated with the DMA subsystem and is accessible using the `dma_alloc_coherent()` DMA API. The CMA can be modified in the kernel configuration, on the kernel command line, or using the `linux,cma-default` DT property that points to the kernel to use reserved memory region as a default CMA memory pool.

Types of DMA mappings

A DMA mapping is a combination of allocating a DMA buffer and generating an address for that buffer that is accessible by the device. There are two types of DMA mappings:

1. **Coherent DMA mappings** use uncached memory mapping from kernel space, usually allocated by using `dma_alloc_coherent()`. The kernel allocates a suitable buffer and sets the mapping for the driver. This memory can simultaneously be accessed by the CPU and the device, so it has to be in a cache coherent memory area, and it is allocated for the whole time the module is loaded. Buffers are usually mapped at driver initialization, unmapped at the end, and to do this the hardware should guarantee that the device and the CPU can access the data in parallel and see updates made by each other without any explicit software flushing.

If the processor has a coherent architecture, the allocator function `dma_alloc_coherent()` will not make the memory uncached. It will only allocate the memory and create a physical to virtual mapping for the CPU.

If the architecture is non-coherent, the `dma_alloc_coherent()` function will make the memory uncached so that coherency is maintained. The `dma_alloc_coherent()` function calls `arm_dma_alloc()`, which in turns calls `__dma_alloc()`, which takes `pgprot_t` as argument, which is basically the page attributes to make this memory uncached.

```
static inline void *dma_alloc_coherent(struct device *dev, size_t size,
                                      dma_addr_t *dma_handle, gfp_t flag)
{
    return dma_alloc_attrs(dev, size, dma_handle, flag, 0);
}

static inline void *dma_alloc_attrs(struct device *dev, size_t size,
                                   dma_addr_t *dma_handle, gfp_t flag,
                                   unsigned long attrs)
{
    struct dma_map_ops *ops = get_dma_ops(dev);
    void *cpu_addr;

    BUG_ON(!ops);

    if (dma_alloc_from_coherent(dev, size, dma_handle, &cpu_addr))
        return cpu_addr;

    if (!arch_dma_alloc_attrs(&dev, &flag))
        return NULL;
    if (!ops->alloc)
        return NULL;

    /* non-coherent architecture, calls arm_dma_alloc() */
    cpu_addr = ops->alloc(dev, size, dma_handle, flag, attrs);
    debug_dma_alloc_coherent(dev, size, *dma_handle, cpu_addr);
    return cpu_addr;
}

/*
 * Allocate DMA-coherent memory space and return both the kernel remapped
 * virtual and bus address for that space.
 */
void *arm_dma_alloc(struct device *dev, size_t size, dma_addr_t *handle,
                   gfp_t gfp, unsigned long attrs)
{
    pgprot_t prot = __get_dma_pgprot(attrs, PAGE_KERNEL);

    return __dma_alloc(dev, size, handle, gfp, prot, false,
                      attrs, __builtin_return_address(0));
}
```

To allocate and map large (`PAGE_SIZE` or so) consistent DMA regions, you should do:

```
#include <linux/dma-mapping.h>
dma_addr_t dma_handle;
cpu_addr = dma_alloc_coherent(dev, size, &dma_handle, gfp);
```


The `dma_alloc_coherent()` function allocates uncached, unbuffered memory for a device for performing DMA. It allocates pages, returns the CPU-viewed (virtual) address and sets the third argument to the device-viewed address. A buffer is automatically placed where the device accesses it. The argument `dev` is a device pointer. This function may be called in an interrupt context with the `GFP_ATOMIC` flag. The `size` argument is the length in bytes of the region you want to allocate, and `gfp` is a standard GFP flag. The `dma_alloc_coherent()` function returns two values: the `cpu_addr` virtual address, which you can use to access it from the CPU and the `dma_handle` DMA address.

The CPU virtual address and the DMA address are both guaranteed to be aligned to the smallest `PAGE_SIZE` order which is greater than or equal to the requested size.

To unmap and free such a DMA region, you will call:

```
dma_free_coherent(dev, size, cpu_addr, dma_handle);
```

Where `dev` and `size` are the same as in the above `dma_alloc_coherent()` call, and `cpu_addr` and `dma_handle` are the values that `dma_alloc_coherent()` returned to you. This function may not be called in interrupt context.

2. **Streaming DMA Mappings** use cached mapping and clean or invalidate it according to the operation using `dma_map_single()` and `dma_unmap_single()`. This is different from coherent mapping because the mapping deals with addresses that were chosen a priori, which are usually mapped for one DMA transfer and unmapped right after it.

For non-coherent processors, the `dma_map_single()` function will call `dma_map_single_attrs()`, which in turn calls `arm_dma_map_page()`, which ensures that any data held in the cache is appropriately discarded or written back.

```
#define dma_map_single(d, a, s, r) dma_map_single_attrs(d, a, s, r, 0)

static inline dma_addr_t dma_map_single_attrs(struct device *dev, void *ptr,
                                              size_t size,
                                              enum dma_data_direction dir,
                                              unsigned long attrs)
{
    struct dma_map_ops *ops = get_dma_ops(dev);
    dma_addr_t addr;

    kmemcheck_mark_initialized(ptr, size);
    BUG_ON(!valid_dma_direction(dir));
    /* calls arm_dma_map_page for ARM architectures */
    addr = ops->map_page(dev, virt_to_page(ptr),
                        offset_in_page(ptr), size,
                        dir, attrs);
    debug_dma_map_page(dev, virt_to_page(ptr),
                      offset_in_page(ptr), size,
                      dir, addr, true);
}
```

```

        return addr;
    }

/*
 * arm_dma_map_page - map a portion of a page for streaming DMA
 * @dev: valid struct device pointer, or NULL for ISA and EISA-like devices
 * @page: page that buffer resides in
 * @offset: offset into page for start of buffer
 * @size: size of buffer to map
 * @dir: DMA transfer direction
 *
 * Ensure that any data held in the cache is appropriately discarded
 * or written back.
 *
 * The device owns this memory once this call has completed. The CPU
 * can regain ownership by calling dma_unmap_page().
 */
static dma_addr_t arm_dma_map_page(struct device *dev, struct page *page,
                                   unsigned long offset, size_t size, enum dma_data_direction dir,
                                   unsigned long attrs)
{
    if ((attrs & DMA_ATTR_SKIP_CPU_SYNC) == 0)
        __dma_page_cpu_to_dev(page, offset, size, dir);
    return pfn_to_dma(dev, page_to_pfn(page)) + offset;
}

```

The streaming DMA mapping routines can be called from interrupt context. There are two versions of each map/unmap, one that will map/unmap a single memory region, and one that will map/unmap a scatterlist.

To map a single region, see the code snippet below:

```

struct device *dev = &my_dev->dev;
dma_addr_t dma_handle;
void *addr = buffer->ptr;
size_t size = buffer->len;
dma_handle = dma_map_single(dev, addr, size, direction);

```

Where `dev` is a device pointer, `addr` is the pointer that contains the virtual buffer address allocated with `kmalloc()`, `size` is the buffer size, and the direction choices are: `DMA_BIDIRECTIONAL`, `DMA_TO_DEVICE` or `DMA_FROM_DEVICE`. The `dma_handle` is the returned DMA bus address.

To unmap the memory region, you will use the following function:

```

dma_unmap_single(dev, dma_handle, size, direction);

```

You should call `dma_unmap_single()` when the DMA activity is finished, e.g., from the interrupt service routine that indicated that the DMA transfer is done.

These are the rules for streaming DMA mapping:

- A buffer can only be used in the direction specified.
- A mapped buffer belongs to the device, not the processor. The device driver must keep its hands off the buffer until it is unmapped.
- A buffer used to send data to a device must contain the data before it is mapped.
- The buffer must not be unmapped while DMA is still active, or serious system instability is guaranteed.

LAB 9.1: "streaming DMA" module

In this LAB 9.1, you will develop a DMA driver that uses the streaming DMA mapping. This driver will allocate two kernel buffers named `wbuf` and `rbuf`. The driver will receive characters from user space and store them in the `wbuf` buffer, then it will set up a DMA transaction (memory to memory) to copy buffer data from `wbuf` to `rbuf`. Finally, both buffers will be compared to check if they contain the same values or not.

The main code sections of the driver will now be described:

1. Include the required header files:

```
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/dma-mapping.h> /* DMA mapping functions */
#include <linux/fs.h>

/* Functions needed to allocate a DMA slave channel, set slave and controller
 * specific parameters, get a descriptor for transaction, submit the
 * transaction, issue pending requests and wait for callback notification
 */
#include <linux/dmaengine.h>
#include <linux/miscdevice.h>
#include <linux/platform_device.h>
#include <linux/of_device.h>
```

2. Create a private structure that will store the DMA device-specific information. In this driver, you will handle a char device, so a `miscdevice` structure will be created, initialized and added to your private structure in its first field. The `wbuf` and `rbuf` pointer variables will hold the addresses of your allocated buffers. The `dma_m2m_chan` pointer variable will hold the DMA channel associated with the dev device.

```
struct dma_private
{
    struct miscdevice dma_misc_device;
    struct device *dev;
    char *wbuf;
    char *rbuf;
    struct dma_chan *dma_m2m_chan;
```

```
    struct completion dma_m2m_ok;
};
```

The last field of your private structure is a completion variable. A common pattern in kernel programming involves initiating some activity outside of the current thread, then waiting for that activity to complete. This activity can be the creation of a new kernel thread, a request to an existing process, or some sort of hardware-based action (like a DMA transfer). In such cases, it can be tempting to use a semaphore to synchronize both tasks, as shown in the following code snippet:

```
struct semaphore sem;
init_MUTEX_LOCKED(&sem);
start_external_task(&sem);
down(&sem);
```

The external task can then call `up(&sem)` when its work is done. As it turns out, semaphores are not the best tool to use in this situation. In normal use, code attempting to lock a semaphore finds that semaphore is available almost all the time; if there is significant contention for the semaphore, performance suffers, and the locking scheme needs to be reviewed. When the task completion is communicated in the way shown above, the thread calling `down` will almost always have to wait; performance will suffer accordingly. Semaphores can also be subject to a (difficult) race condition when used in this way if they are declared as automatic variables. In some cases, the semaphore could vanish before the process calling `up` is finished with it.

These concerns inspired the addition of the completion interface in the 2.4.7 kernel.

Completions are a lightweight mechanism with one task: allowing one thread to tell another that the job is done. The advantage of using completions is to generate more efficient code, as both threads can continue until the result is actually needed.

3. In the `probe()` function, set up the capabilities for the channel that will be requested, allocate the `wbuf` and `rbuf` buffers, and request the DMA channel from the DMA engine by using the `dma_request_channel()` function. The `dma_request_channel()` function takes three parameters:
 - The `dma_m2m_mask` that holds the channel capabilities.
 - The `m2m_dma_data` custom data structure (not needed in our driver).
 - The `dma_m2m_filter` that helps to select a more specific channel between multiple channel possibilities. When allocating a channel, the dma engine finds the first channel that matches the mask and calls the filter function.

See below a code snippet of our driver's `probe()` function:

```
static int my_probe(struct platform_device *pdev)
{
    int retval;
    struct dma_private *dma_device;
    dma_cap_mask_t dma_m2m_mask;

    dma_device = devm_kzalloc(&pdev->dev, sizeof(struct dma_private), GFP_KERNEL);

    dma_device->dma_misc_device.minor = MISC_DYNAMIC_MINOR;
    dma_device->dma_misc_device.name = "sdma_test";
    dma_device->dma_misc_device.fops = &dma_fops;

    dma_device->dev = &pdev->dev;

    dma_device->wbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);

    dma_device->rbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);

    dma_cap_zero(dma_m2m_mask);
    dma_cap_set(DMA_MEMCPY, dma_m2m_mask);
    dma_device->dma_m2m_chan = dma_request_channel(dma_m2m_mask, 0, NULL);

    misc_register(&dma_device->dma_misc_device);

    platform_set_drvdata(pdev, dma_device);

    return 0;
}
```

4. Write the `sdma_write()` function to communicate with user space. This function gets the characters written to the char device by using `copy_from_user()` and store them in the `wbuf` buffer. The `dma_src` and `dma_dst` DMA addresses are obtained by using the `dma_map_single()` function, which takes as parameters the `wbuf` and `rbuf` virtual addresses previously obtained in the `probe()` function and stored in your DMA private structure. These virtual addresses are retrieved in `sdma_write()` by using the `container_of()` function.

Get a descriptor for the transaction using `device_prep_dma_memcpy()`. Once the descriptor has been obtained, the callback information can be added, and the descriptor must be submitted by using `dmaengine_submit()`.

The `dmaengine_submit()` function will not start the DMA operation, it merely adds it to the pending queue. For this, do `dma_async_issue_pending()`. The transactions in the pending queue can be activated by calling the `issue_pending` API. If the channel is idle, then the first transaction in the queue is started and subsequent ones queued up. On completion of each DMA operation, the next in queue is started and a tasklet triggered. The tasklet will then call the client driver's completion callback routine for notification.

```

static ssize_t sdma_write(struct file * file, const char __user * buf,
                        size_t count, loff_t * offset)
{
    struct dma_async_tx_descriptor *dma_m2m_desc;
    struct dma_device *dma_dev;
    struct dma_private *dma_priv;
    dma_cookie_t cookie;
    dma_addr_t dma_src;
    dma_addr_t dma_dst;

    dma_priv = container_of(file->private_data, struct dma_private, dma_misc_device);

    dma_dev = dma_priv->dma_m2m_chan->device;

    if(copy_from_user(dma_priv->wbuf, buf, count)) {
        return -EFAULT;
    }

    dev_info(dma_priv->dev, "The wbuf string is %s\n", dma_priv->wbuf);

    dma_src = dma_map_single(dma_priv->dev, dma_priv->wbuf,
                            SDMA_BUF_SIZE, DMA_TO_DEVICE);

    dev_info(dma_priv->dev, "dma_src map obtained");

    dma_dst = dma_map_single(dma_priv->dev, dma_priv->rbuf,
                            SDMA_BUF_SIZE, DMA_TO_DEVICE);

    dev_info(dma_priv->dev, "dma_dst map obtained");

    dma_m2m_desc = dma_dev->device_prep_dma_memcpy(dma_priv->dma_m2m_chan,
                                                    dma_dst,
                                                    dma_src,
                                                    SDMA_BUF_SIZE,
                                                    DMA_CTRL_ACK | DMA_PREP_INTERRUPT);

    dev_info(dma_priv->dev, "successful descriptor obtained");

    dma_m2m_desc->callback = dma_m2m_callback;
    dma_m2m_desc->callback_param = dma_priv;
    init_completion(&dma_priv->dma_m2m_ok);

    cookie = dmaengine_submit(dma_m2m_desc);

    if (dma_submit_error(cookie)){
        dev_err(dma_priv->dev, "Failed to submit DMA\n");
        return -EINVAL;
    };
    dma_async_issue_pending(dma_priv->dma_m2m_chan);
    wait_for_completion(&dma_priv->dma_m2m_ok);
    dma_async_is_tx_complete(dma_priv->dma_m2m_chan, cookie, NULL, NULL);

    dev_info(dma_priv->dev, "The rbuf string is %s\n", dma_priv->rbuf);
}

```

```

        dma_unmap_single(dma_priv->dev, dma_src, SDMA_BUF_SIZE, DMA_TO_DEVICE);
        dma_unmap_single(dma_priv->dev, dma_dst, SDMA_BUF_SIZE, DMA_TO_DEVICE);

        return count;
}

```

5. Create a callback function to inform about the completion of the DMA transaction. Signal the completion of the event inside this function:

```

static void dma_m2m_callback(void *data)
{
    struct dma_private *dma_priv = data;
    dev_info(dma_priv->dev, "%s\n finished DMA transaction" ,__func__);
    complete(&dma_priv->dma_m2m_ok);

    if (*(dma_priv->rbuf) != *(dma_priv->wbuf))
        dev_err(dma_priv->dev, "buffer copy failed!\n");

    dev_info(dma_priv->dev, "buffer copy passed!\n");
    dev_info(dma_priv->dev, "wbuf is %s\n", dma_priv->wbuf);
    dev_info(dma_priv->dev, "rbuf is %s\n", dma_priv->rbuf);
}

```

6. Open the `bcm2710-rpi-3-b.dts` DT file, and add the `sdma_m2m` node in the soc node:

```

&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };

    [...]

    sdma_m2m {
        compatible = "arrow,sdma_m2m";
    };
};

```

7. Create a new `sdma_rpi3_m2m.c` file in the `linux_5.4_rpi3_drivers` folder, and add `sdma_rpi3_m2m.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```

~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy

```

8. Build the modified Device Tree, and load it to the target processor:

```

~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/

```

9. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 9-1: sdma_rpi3_m2m.c

```
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/dma-mapping.h>
#include <linux/fs.h>
#include <linux/dmaengine.h>
#include <linux/miscdevice.h>
#include <linux/platform_device.h>
#include <linux/of_device.h>

struct dma_private
{
    struct miscdevice dma_misc_device;
    struct device *dev;
    char *wbuf;
    char *rbuf;
    struct dma_chan *dma_m2m_chan;
    struct completion dma_m2m_ok;
};

#define SDMA_BUF_SIZE 4096

static void dma_m2m_callback(void *data)
{
    struct dma_private *dma_priv = data;
    dev_info(dma_priv->dev, "%s\n finished DMA transaction", __func__);
    complete(&dma_priv->dma_m2m_ok);

    if (*(dma_priv->rbuf) != *(dma_priv->wbuf))
        dev_err(dma_priv->dev, "buffer copy failed!\n");

    dev_info(dma_priv->dev, "buffer copy passed!\n");
    dev_info(dma_priv->dev, "wbuf is %s\n", dma_priv->wbuf);
    dev_info(dma_priv->dev, "rbuf is %s\n", dma_priv->rbuf);
}

static ssize_t sdma_write(struct file * file, const char __user * buf,
                        size_t count, loff_t * offset)
{
    struct dma_async_tx_descriptor *dma_m2m_desc;
    struct dma_device *dma_dev;
    struct dma_private *dma_priv;
    dma_cookie_t cookie;
    dma_addr_t dma_src;
    dma_addr_t dma_dst;

    dma_priv = container_of(file->private_data, struct dma_private, dma_misc_device);
```



```

dma_dev = dma_priv->dma_m2m_chan->device;

if(copy_from_user(dma_priv->wbuf, buf, count)) {
    return -EFAULT;
}

dev_info(dma_priv->dev, "The wbuf string is %s\n", dma_priv->wbuf);

dma_src = dma_map_single(dma_priv->dev, dma_priv->wbuf, SDMA_BUF_SIZE, DMA_TO_DEVICE);
dev_info(dma_priv->dev, "dma_src map obtained");

dma_dst = dma_map_single(dma_priv->dev, dma_priv->rbuf, SDMA_BUF_SIZE, DMA_TO_DEVICE);
dev_info(dma_priv->dev, "dma_dst map obtained");

dma_m2m_desc = dma_dev->device_prep_dma_memcpy(dma_priv->dma_m2m_chan,
                                                dma_dst,
                                                dma_src,
                                                SDMA_BUF_SIZE,
                                                DMA_CTRL_ACK | DMA_PREP_INTERRUPT);

dev_info(dma_priv->dev, "successful descriptor obtained");

dma_m2m_desc->callback = dma_m2m_callback;
dma_m2m_desc->callback_param = dma_priv;
init_completion(&dma_priv->dma_m2m_ok);

cookie = dmaengine_submit(dma_m2m_desc);

if (dma_submit_error(cookie)){
    dev_err(dma_priv->dev, "Failed to submit DMA\n");
    return -EINVAL;
};
dma_async_issue_pending(dma_priv->dma_m2m_chan);
wait_for_completion(&dma_priv->dma_m2m_ok);
dma_async_is_tx_complete(dma_priv->dma_m2m_chan, cookie, NULL, NULL);

dev_info(dma_priv->dev, "The rbuf string is %s\n", dma_priv->rbuf);

dma_unmap_single(dma_priv->dev, dma_src, SDMA_BUF_SIZE, DMA_TO_DEVICE);
dma_unmap_single(dma_priv->dev, dma_dst, SDMA_BUF_SIZE, DMA_TO_DEVICE);

return count;
}

struct file_operations dma_fops = {
    write:  sdma_write,
};

static int my_probe(struct platform_device *pdev)
{
    int retval;

```

```

struct dma_private *dma_device;
dma_cap_mask_t dma_m2m_mask;

dev_info(&pdev->dev, "platform_probe enter\n");

dma_device = devm_kzalloc(&pdev->dev, sizeof(struct dma_private), GFP_KERNEL);

dma_device->dma_misc_device.minor = MISC_DYNAMIC_MINOR;
dma_device->dma_misc_device.name = "sdma_test";
dma_device->dma_misc_device.fops = &dma_fops;

dma_device->dev = &pdev->dev;

dma_device->wbuff = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
if(!dma_device->wbuff) {
    dev_err(&pdev->dev, "error allocating wbuff !!\n");
    return -ENOMEM;
}

dma_device->rbuf = devm_kzalloc(&pdev->dev, SDMA_BUF_SIZE, GFP_KERNEL);
if(!dma_device->rbuf) {
    dev_err(&pdev->dev, "error allocating rbuf !!\n");
    return -ENOMEM;
}

dma_cap_zero(dma_m2m_mask);
dma_cap_set(DMA_MEMCPY, dma_m2m_mask);
dma_device->dma_m2m_chan = dma_request_channel(dma_m2m_mask, 0, NULL);
if (!dma_device->dma_m2m_chan) {
    dev_err(&pdev->dev, "Error opening the SDMA memory to memory channel\n");
    return -EINVAL;
}

retval = misc_register(&dma_device->dma_misc_device);
if (retval) return retval;

platform_set_drvdata(pdev, dma_device);

dev_info(&pdev->dev, "platform_probe exit\n");

return 0;
}

static int my_remove(struct platform_device *pdev)
{
    struct dma_private *dma_device = platform_get_drvdata(pdev);
    dev_info(&pdev->dev, "platform_remove enter\n");
    misc_deregister(&dma_device->dma_misc_device);
    dma_release_channel(dma_device->dma_m2m_chan);
    dev_info(&pdev->dev, "platform_remove exit\n");
    return 0;
}

static const struct of_device_id my_of_ids[] = {

```

```

    { .compatible = "arrow,sdma_m2m"},
    {}},
};
MODULE_DEVICE_TABLE(of, my_of_ids);

static struct platform_driver my_platform_driver = {
    .probe = my_probe,
    .remove = my_remove,
    .driver = {
        .name = "sdma_m2m",
        .of_match_table = my_of_ids,
        .owner = THIS_MODULE,
    }
};

static int demo_init(void)
{
    int ret_val;
    pr_info("demo_init enter\n");

    ret_val = platform_driver_register(&my_platform_driver);
    if (ret_val != 0)
    {
        pr_err("platform value returned %d\n", ret_val);
        return ret_val;
    }
    pr_info("demo_init exit\n");
    return 0;
}

static void demo_exit(void)
{
    pr_info("demo_exit enter\n");
    platform_driver_unregister(&my_platform_driver);
    pr_info("demo_exit exit\n");
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a SDMA memory to memory driver");

```

sdma_rpi3_m2m.ko demonstration

Load the module:

```
root@raspberrypi:/home/pi# insmod sdma_rpi3_m2m.ko
demo_init enter
sdma_m2m soc:sdma_m2m: platform_probe enter
sdma_m2m soc:sdma_m2m: platform_probe exit
demo_init exit
```

Write values to the wbuf buffer, then start a DMA transaction which copies values from wbuf to rbuf. Finally, compare both buffers values:

```
root@raspberrypi:/home/pi# echo abcdefg > /dev/sdma_test
sdma_m2m soc:sdma_m2m: The wbuf string is abcdefg
sdma_m2m soc:sdma_m2m: dma_src map obtained
sdma_m2m soc:sdma_m2m: dma_dst map obtained
sdma_m2m soc:sdma_m2m: successful descriptor obtained
sdma_m2m soc:sdma_m2m: dma_m2m_callback
finished DMA transaction
sdma_m2m soc:sdma_m2m: buffer copy passed!
sdma_m2m soc:sdma_m2m: The rbuf string is abcdefg
sdma_m2m soc:sdma_m2m: wbuf is abcdefg
sdma_m2m soc:sdma_m2m: rbuf is abcdefg
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmmod sdma_rpi3_m2m.ko
demo_exit enter
sdma_m2m soc:sdma_m2m: platform_remove enter
sdma_m2m soc:sdma_m2m: platform_remove exit
demo_exit exit
```

10

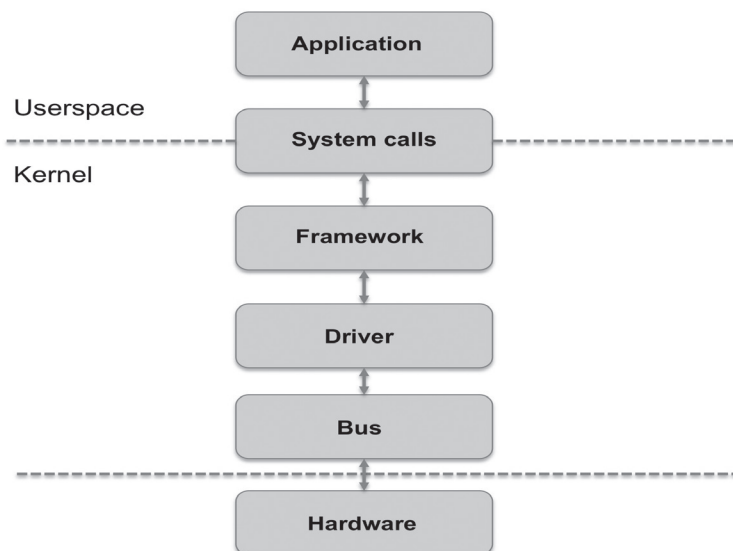
Input Subsystem

Many Linux device drivers are not implemented directly as character drivers. They are implemented under a framework, specific to a given device type (e.g., networking, MTD, RTC, v4L2, serial, IIO). The framework factors out the common parts of drivers for the same type of devices to reduce code duplication.

The Linux frameworks allow the provision of a coherent user space interface for every type of device, regardless of the driver. The application can still see many of the device drivers as character devices. For example, the Linux network subsystem provides a socket API such that an application can connect to a network using any network driver without knowing the details of the network driver.

Throughout this chapter, you will explore the Input subsystem in detail. You will also develop several drivers that will help you to understand the use of this framework.

Observe in the following image how the driver interfaces with a framework (to expose the hardware to the user application) and with a bus infrastructure (part of the device model that communicates with the hardware):



Input subsystem drivers

The Input subsystem takes care of all the input events coming from the human user. Input device drivers will capture the hardware event information in a uniform format (`input_event` structure) and report to the core layer, then the core layer sorts the data, escalates to the appropriate event-handling driver, and finally, through the event layer, passes the information to user space. Applications using `/dev` device node can get event information.

Initially written to support the USB HID (Human Interface Device) devices, the Input subsystem quickly grew up to handle all types of inputs (using USB or not): keyboards, mice, joysticks, touchscreens, etc.

The Input subsystem is split in two parts:

1. **Device drivers:** They talk to the hardware (e.g., USB, I2C) and provide events (e.g., keystrokes, accelerometer movements, touchscreen coordinates) to the input core.
2. **Event handlers:** Input event drivers that get events from device drivers and pass them to user space and in-kernel consumers, as needed via various interfaces. The **evdev** driver is a generic input event interface in the Linux kernel. It generalizes raw input events from

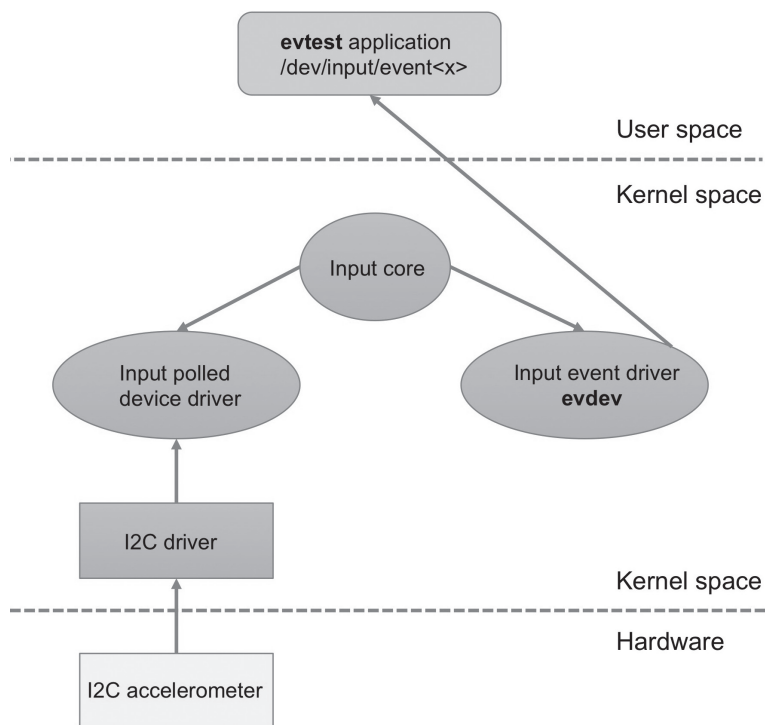
device drivers and makes them available through character devices in the `/dev/input/` directory. The event interface will represent each input device as a `/dev/input/event<X>` character device. This is the preferred interface for user space to consume user input, and all clients are encouraged to use it.

You can use blocking and nonblocking reads and also `select()` on the `/dev/input/eventX` devices, and you'll always get a whole number of input events on a read. Their layout is:

```
struct input_event {  
    struct timeval time;  
    unsigned short type;  
    unsigned short code;  
    unsigned int value;  
};
```

A very useful application for input device testing is `evtest`, which is located at <http://cgib.freedesktop.org/evtest/>. The `evtest` application displays information on the input device specified on the command line, including all the events supported by the device.

In the next figure, you can see an Input subsystem diagram that can be used as an example for the next kernel module lab, where you will control an I2C accelerometer by using the Input subsystem.



LAB 10.1: "input subsystem accelerometer" module

In this kernel module, you will control the tilt of an accelerometer board connected to the I2C bus of the Raspberry Pi device. You will use the ADXL345 Accel click mikroBUS™ accessory board to develop the driver. You can access to the schematics of the board at <http://www.mikroe.com/click/accel/>, or you can download the schematics from the GitHub of this book.

Your driver will scan periodically the value of one of the accelerometer axes, and depending on the tilt of the board, it will generate an event that is exposed to the application **evtest**.

In this accelerometer kernel module, you will use the **polled input** subclass. A polled input device provides a skeleton for supporting simple input devices that do not raise interrupts, but have to be periodically scanned or polled to detect changes in their state.

A polled input device is described by the `input_polled_dev` structure, declared in `include/linux/input-polldev.h` in the kernel source tree:

```
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;

/* private: */
struct delayed_work work;

bool devres_managed;
};

```

You will allocate and free struct `input_polled_dev` by using the following functions:

```

struct input_polled_dev *input_allocate_polled_device(void);
void input_free_polled_device(struct input_polled_dev *dev);

```

The accelerometer driver will support `EV_KEY` type events, with a `KEY_1` event that will be set to 0 or 1 depending on the board's tilt. The `set_bit()` call is an atomic operation, allowing it to set a particular bit to 1.

```

set_bit(EV_KEY, ioaccel->polled_input->input->evbit); /* supported event types (support for
EV_KEY events) */
set_bit(KEY_1, ioaccel->polled_input->input->keybit); /* Set the event code support (event
KEY_1 ) */

```

The `input_polled_dev` structure will be handled by the `poll()` callback function. This function polls the device and posts input events. The `poll_interval` field will be set to 50 ms in your driver. Inside the `poll()` function, the event is sent to the event handler by using the `input_event()` function.

After submitting the event, the input core must be notified by using the `input_sync()` function:

```

void input_sync(struct input_dev *dev);

```

The device registration/unregistration is done with the following functions:

```

int input_register_polled_device(struct input_polled_dev *dev);
void input_unregister_polled_device(struct input_polled_dev *dev);

```

The main code sections of the driver will be described using three categories: Device Tree, Input framework as an I2C interaction, and Input framework as an input device.

You will use the Raspberry Pi GPIO expansion connector to obtain the I2C signals. The GPIO2 and GPIO3 pins will be used to get the SDA1 and SCL1 signals. Connect them to the pins SDA and SCL of the ADXL345 Accel click mikroBUS™ accessory board. Do not forget to connect 3.3V and GND between the two boards.

Device Tree

Open the `bcm2710-rpi-3-b.dts` DT file, and add the `adxl345@1c` node in the `i2c1` controller master node. The `reg` property provides the ADXL345 I2C address:

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
    status = "okay";

    [...]

    adxl345@1c {
        compatible = "arrow,adxl345";
        reg = <0x1d>;
    };
};
```

Build the modified Device Tree, and load it to the Raspberry Pi:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Input framework as an I2C interaction

These are the main code sections:

1. Include the required header files:

```
#include <linux/i2c.h> /* struct i2c_driver, struct i2c_client(), i2c_get_clientdata(),
i2c_set_clientdata() */
```

2. Create an `i2c_driver` structure:

```
static struct i2c_driver ioaccel_driver = {
    .driver = {
        .name = "adxl345",
        .owner = THIS_MODULE,
        .of_match_table = ioaccel_dt_ids,
    },
    .probe = ioaccel_probe,
    .remove = ioaccel_remove,
    .id_table = i2c_ids,
};
```

3. Register to the I2C bus as a driver:

```
module_i2c_driver(ioaccel_driver);
```

4. Add "adx1345" to the list of devices supported by the driver:

```
static const struct of_device_id ioaccel_dt_ids[] = {
    { .compatible = "arrow,adx1345", },
    { }
};
MODULE_DEVICE_TABLE(of, ioaccel_dt_ids);
```

5. Define an array of i2c_device_id structures:

```
static const struct i2c_device_id i2c_ids[] = {
    { "adx1345", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, i2c_ids);
```

6. Use SMBus functions for accessing to the accelerometer registers. After VS is applied, the ADXL345 device enters standby mode, where power consumption is minimized, and the device waits for VDD I/O to be applied and for the command to enter measurement mode to be received. This command can be initiated by setting the measure bit (Bit D3) in the POWER_CTL register (Address 0x2D):

```
#define POWER_CTL      0x2D
#define PCTL_MEASURE    (1 << 3)
#define OUT_X_MSB      0x33

/* Enter measurement mode */
i2c_smbus_write_byte_data(client, POWER_CTL, PCTL_MEASURE);
```

The axis value is read with the following line of code:

```
i2c_smbus_read_byte_data(ioaccel->i2c_client, OUT_X_MSB);
```

Input framework as an input device

These are the main code sections:

1. Include the required header files:

```
# include linux/input-polldev.h /* struct input_polled_dev, input_allocate_polled_device(),
input_register_polled_device() */
```

2. The device model needs to keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices (devices handled by subsystems, like the Input subsystem in this case). This need is typically implemented by creating a private data structure to manage the device and implement such pointers between the physical and logical worlds. As you have seen in other labs throughout this book, this private structure allows the driver to manage multiple devices using the same driver. Add the following private structure definition to your driver code:

```
struct ioaccel_dev {
    struct i2c_client *i2c_client;
    struct input_polled_dev *polled_input;
};
```

3. In the `ioaccel_probe()` function, declare an instance of this structure, and allocate it using `devm_kzalloc()`:

```
struct ioaccel_dev *ioaccel;
ioaccel = devm_kzalloc(&client->dev, sizeof(struct ioaccel_dev), GFP_KERNEL);
```

4. To be able to access your private data structure in other functions of the driver, you need to attach it to the `i2c_client` structure by using the `i2c_set_clientdata()` function. This function stores `ioaccel` in `client->dev->driver_data`. You can retrieve the `ioaccel` pointer from the private structure using the function `i2c_get_clientdata(client)`:

```
i2c_set_clientdata(client, ioaccel); /* Write it in the probe() function */
ioaccel = i2c_get_clientdata(client); /* Write it in the remove() function */
```

5. Allocate the `input_polled_dev` structure in `probe()` by using the following line of code:

```
ioaccel->polled_input = devm_input_allocate_polled_device(&client->dev);
```

6. Initialize the polled input device. Keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices:

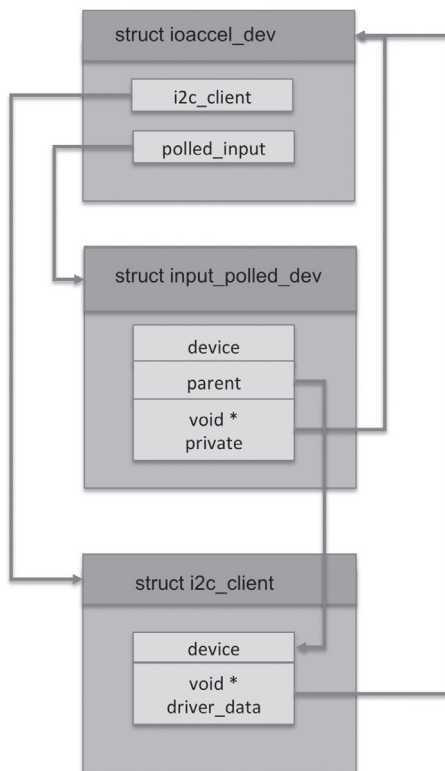
```
ioaccel->i2c_client = client; /* Keep pointer to the I2C device, needed for exchanging data with
the accelerometer */
ioaccel->polled_input->private = ioaccel; /* struct polled_input can store the driver-specific
data in void *private. Place the pointer to the private structure here; in this way, you will be
able to recover the ioaccel pointer later (as it can be seen for example in the ioaccel_poll()
function) */
ioaccel->polled_input->poll_interval = 50; /* Callback interval */
```

```

ioaccel->polled_input->poll = ioaccel_poll; /* Callback that will be called every 50 ms interval
*/
ioaccel->polled_input->input->parent = &client->dev; /* Keep pointers between physical
devices and logical devices */
ioaccel->polled_input->input->name = "IOACCEL keyboard"; /* Input sub-device parameters that
will appear in log on registering the device */
ioaccel->polled_input->input->id.bustype = BUS_I2C; /* Input sub-device parameters */

```

See the links between physical and logical device structures in the next figure:



7. Set the event type and the event generated for this device:

```

set_bit(EV_KEY, ioaccel->polled_input->input->evbit); /* Supported event type (support
for EV_KEY events) */
set_bit(KEY_1, ioaccel->polled_input->input->keybit); /* Set the event code support
(event KEY_1 ) */

```

8. Register in `probe()` and unregister in `remove()` the `polled_input` device to the input core. Once registered, the device is global for the rest of the driver functions until it is unregistered. After this call, the device is ready to accept requests from user space applications.

```
input_register_polled_device(ioaccel->polled_input);
input_unregister_polled_device(ioaccel->polled_input);
```

9. Write the `ioaccel_poll()` function. This function will be called every 50ms to read the `OUT_X_MSB` register (address `0x33`) of the ADXL345 accelerometer by using the `i2c_smbus_read_byte_data()` function. The first parameter of the `i2c_smbus_read_byte_data()` function is a pointer to the `i2c_client` structure. This pointer will allow you to get the ADXL345 I2C address (`0x1d`). The `0x1d` value will be retrieved from `client->address`. After binding, the I2C bus driver gets this I2C address value from the `ioaccel` Device Tree node and stores it in the `i2c_client` structure, then this I2C address is sent to the `ioaccel_probe()` function via a client pointer variable that points to this `i2c_client` structure.

An input event `KEY_1` will be reported with values of 0 or 1, depending on the ADXL345 board's tilt. You can use a different range of acceleration values to report these events.

```
static void ioaccel_poll(struct input_polled_dev * pl_dev)
{
    struct ioaccel_dev * ioaccel = pl_dev->private;
    int val = 0;
    val = i2c_smbus_read_byte_data(ioaccel->i2c_client, OUT_X_MSB);

    if ( (val > 0xc0) && (val < 0xff) ) {
        input_event(ioaccel->polled_input->input, EV_KEY, KEY_1, 1);
    } else {
        input_event(ioaccel->polled_input->input, EV_KEY, KEY_1, 0);
    }

    input_sync(ioaccel->polled_input->input);
}
```

10. Create a new `i2c_rpi3_accel.c` file in the `linux_5.4_rpi3_drivers` folder, and add `i2c_rpi3_accel.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

Listing 10-1: i2c_rpi3_accel.c

```

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/i2c.h>
#include <linux/input-polldev.h>

/* Create private structure */
struct ioaccel_dev {
    struct i2c_client *i2c_client;
    struct input_polled_dev *polled_input;
};

#define POWER_CTL            0x2D
#define PCTL_MEASURE         (1 << 3)
#define OUT_X_MSB            0x33

/* Poll function */
static void ioaccel_poll(struct input_polled_dev * pl_dev)
{
    struct ioaccel_dev *ioaccel = pl_dev->private;
    int val = 0;
    val = i2c_smbus_read_byte_data(ioaccel->i2c_client, OUT_X_MSB);

    if ( (val > 0xc0) && (val < 0xff) ) {
        input_event(ioaccel->polled_input->input, EV_KEY, KEY_1, 1);
    } else {
        input_event(ioaccel->polled_input->input, EV_KEY, KEY_1, 0);
    }

    input_sync(ioaccel->polled_input->input);
}

static int ioaccel_probe(struct i2c_client * client, const struct i2c_device_id * id)
{
    /* Declare an instance of the private structure */
    struct ioaccel_dev *ioaccel;

    dev_info(&client->dev, "my_probe() function is called.\n");

    /* Allocate private structure for new device */
    ioaccel = devm_kzalloc(&client->dev, sizeof(struct ioaccel_dev), GFP_KERNEL);

    /* Associate client->dev with ioaccel private structure */
    i2c_set_clientdata(client, ioaccel);

    /* Enter measurement mode */
    i2c_smbus_write_byte_data(client, POWER_CTL, PCTL_MEASURE);

    /* Allocate struct input_polled_dev */
    ioaccel->polled_input = devm_input_allocate_polled_device(&client->dev);

    /* Initialize polled input */

```

```

ioaccel->i2c_client = client;
ioaccel->polled_input->private = ioaccel;
ioaccel->polled_input->poll_interval = 50;
ioaccel->polled_input->poll = ioaccel_poll;
ioaccel->polled_input->input->dev.parent = &client->dev;
ioaccel->polled_input->input->name = "IOACCEL keyboard";
ioaccel->polled_input->input->id.bustype = BUS_I2C;

/* Set event types */
set_bit(EV_KEY, ioaccel->polled_input->input->evbit);
set_bit(KEY_1, ioaccel->polled_input->input->keybit);

/* Register the device, now the device is global until being unregistered */
input_register_polled_device(ioaccel->polled_input);

return 0;
}

static int ioaccel_remove(struct i2c_client * client)
{
    struct ioaccel_dev *ioaccel;
    ioaccel = i2c_get_clientdata(client);
    input_unregister_polled_device(ioaccel->polled_input);
    dev_info(&client->dev, "ioaccel_remove()\n");
    return 0;
}

/* Add entries to Device Tree */
static const struct of_device_id ioaccel_dt_ids[] = {
    { .compatible = "arrow,adx1345", },
    { }
};
MODULE_DEVICE_TABLE(of, ioaccel_dt_ids);

static const struct i2c_device_id i2c_ids[] = {
    { "adx1345", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, i2c_ids);

/* Create struct i2c_driver */
static struct i2c_driver ioaccel_driver = {
    .driver = {
        .name = "adx1345",
        .owner = THIS_MODULE,
        .of_match_table = ioaccel_dt_ids,
    },
    .probe = ioaccel_probe,
    .remove = ioaccel_remove,
    .id_table = i2c_ids,
};

/* Register to i2c bus as a driver */
module_i2c_driver(ioaccel_driver);

```



```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is an accelerometer INPUT framework platform driver");
```

i2c_rpi3_accel.ko demonstration

Scan the I2C bus with the i2c-tools suite. List the available I2C buses and the devices connected to the i2c-1 bus:

```
root@raspberrypi:/home/pi# i2cdetect -l
i2c-1  i2c      bcm2835 (i2c@7e804000)      I2C adapter
```

```
root@raspberrypi:/home/pi# i2cdetect -y 1
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  1d  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```

Set measurement mode for the ADXL345 device:

```
root@raspberrypi:/home/pi# i2cset -y 1 0x1d 0x2d 0x08
```

The 0x33 value corresponds to the OUT_X_MSB register address. Move the Raspberry Pi board, and see how changes the OUT_X_MSB register value. Set a range of values to generate an event. This range of values will be set inside the ioaccel_poll() function:

```
root@raspberrypi:/home/pi# while true; do i2cget -y 1 0x1d 0x33; done
```

Load the module:

```
root@raspberrypi:/home/pi# insmod i2c_rpi3_accel.ko
adxl345 1-001d: my_probe() function is called.
input: IOACCEL keyboard as /devices/platform/soc/3f804000.i2c/i2c
-1/1-001d/input/input0
```

Execute the evtest application to see the input devices available. Select 0. Move the Raspberry Pi board until the event EV_KEY is generated:

```
root@raspberrypi:/home/pi# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      IOACCEL keyboard
Select the device event number [0-0]: 0
Input driver version is 1.0.1
Input device ID: bus 0x18 vendor 0x0 product 0x0 version 0x0
Input device name: "IOACCEL keyboard"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 2 (KEY_1)
```

Properties:

Testing ... (interrupt to exit)

```
Event: time 1617904839.834963, type 1 (EV_KEY), code 2 (KEY_1), value 1
Event: time 1617904839.834963, ----- SYN_REPORT -----
Event: time 1617904839.895074, type 1 (EV_KEY), code 2 (KEY_1), value 0
Event: time 1617904839.895074, ----- SYN_REPORT -----
Event: time 1617904842.054966, type 1 (EV_KEY), code 2 (KEY_1), value 1
Event: time 1617904842.054966, ----- SYN_REPORT -----
Event: time 1617904842.115023, type 1 (EV_KEY), code 2 (KEY_1), value 0
Event: time 1617904842.115023, ----- SYN_REPORT -----
```

Exit with ^C:

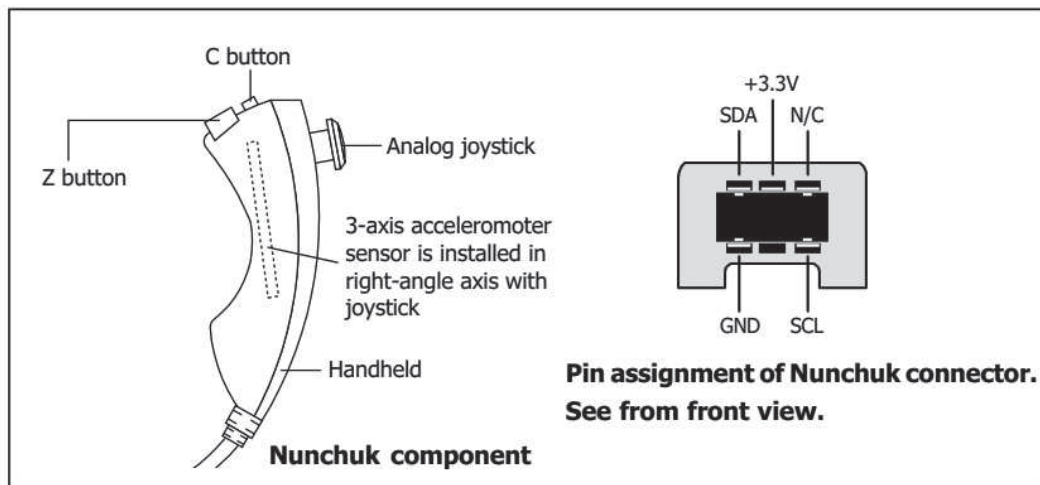
```
root@raspberrypi:/home/pi#
```

Remove the module:

```
root@raspberrypi:/home/pi# rmmod i2c_rpi3_accel.ko
```

LAB 10.2: “Nunchuk input subsystem” module

In this lab, you will develop a driver to control the **WII Nunchuk**. The Wii Nunchuk is a controller for the Wii game console. It includes a 3-axis accelerometer, a X-Y Joystick and two buttons. All these features are available through I2C communication. The following image shows the components of the Wii Nunchuk and the pin assignment of the connector.



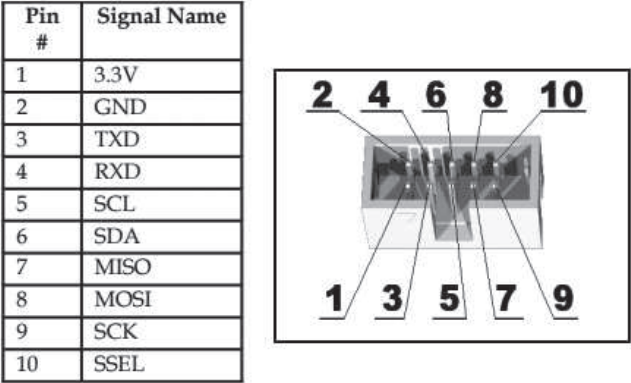
For the development of the lab, you will use the MOD-Wii-UEXT-NUNCHUCK from Olimex: <https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/open-source-hardware>

The Wii Nunchuk uses a proprietary connector with 6 pins, which exposes I2C signals, +3V and Gnd. The MOD-Wii-UEXT-NUNCHUCK includes an UEXT adapter which offers a kind of universal connector supporting three serial communication interfaces: I2C, SPI and RS232. In the MOD-Wii-UEXT-NUNCHUCK, only the I2C signals are connected to the Nunchuk device.

This LAB 10.2 has been inspired by Bootlin’s Embedded Linux kernel and driver development training materials located at <https://bootlin.com/training/kernel/>.

LAB 10.2 hardware description

You will use the I2C pins of the Raspberry Pi 40-pin GPIO header to connect to the Nunchuk’s UEXT connector, which is shown in the following image:



Connect the Raspberry Pi’s I2C pins to the I2C ones of the Nunchuk device:

- Connect Raspberry Pi **SCL** to UEXT **SCL** (Pin 5).
- Connect Raspberry Pi **SDA** to UEXT **SDA** (Pin 6).

Connect the next power pins between the two boards:

- Connect Raspberry Pi **3.3V** to UEXT **3.3V** (Pin 1).
- Connect Raspberry Pi **GND** to UEXT **GND** (Pin 2).

The hardware setup between the two boards is already done!!

LAB 10.2 Device Tree description

Open the `bcm2710-rpi-3-b.dts` DT file and find the `i2c1` controller master node. Inside the `i2c1` node, you can see a `pinctrl-0` property that points to the `i2c1_pins` pin configuration node, which configures the pins of the `i2c1` controller in I2C mode. The `i2c1_pins` pin configuration node is defined in the `bcm2710-rpi-3-b.dts` file, inside the `gpio` node property.

The `i2c1` controller is enabled by writing "okay" to the status property. You will set to 100Khz the clock-frequency property. The Nunchuck device communicates with the Raspberry Pi using an I2C bus interface with a maximum frequency of 100kHz.

You will add the `nunchuk` sub-node to the `i2c1` controller node. There must be a DT device node's compatible property identical to the compatible string stored in one of the driver's `_device_id` structures. The `reg` property includes the I2C address of the device.

You can see below the Device Tree configuration for our Nunchuk device:

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
    status = "okay";

    nunchuk: nunchuk@52 {
        compatible = "nunchuk";
        reg = <0x52>;
    };
};
```

LAB 10.2 Nunchuk controller driver description

The main code sections of the driver will be described using two different categories: I2C driver setup and Input framework driver setup.

I2C driver setup

These are the main code sections:

1. Include the required header files:

```
#include <linux/i2c.h>
```

2. Create an `i2c_driver` structure:

```
static struct i2c_driver nunchuk_driver = {
    .driver = {
        .name = "nunchuk",
        .owner = THIS_MODULE,
        .of_match_table = nunchuk_of_match,
    },
    .probe = nunchuk_probe,
    .remove = nunchuk_remove,
    .id_table = nunchuk_id,
};
```

3. Register to the I2C bus as a driver:

```
module_i2c_driver(nunchuk_driver);
```

4. Add "nunchuk" to the list of devices supported by the driver. The compatible variable matches with the compatible property of the nunchuk DT node:

```
static const struct of_device_id nunchuk_of_match[] = {
    { .compatible = "nunchuk"},
    {}
};
MODULE_DEVICE_TABLE(of, nunchuk_of_match);
```

5. Define an array of `i2c_device_id` structures:

```
static const struct i2c_device_id nunchuk_id[] = {
    {"nunchuk", 0},
    {}
};
MODULE_DEVICE_TABLE(i2c, nunchuk_id);
```

6. You will use the `i2c_master_send()` and `i2c_master_recv()` functions to establish a plain I2C communication with the Nunchuk controller. These routines read/write some bytes from/to a client device. The first parameter of these functions is an I2C client pointer which contains the I2C address of the Nunchuk device (the I2C slave address of the Nunchuk

is 0x52). The second parameter is the buffer to read/write, and the third parameter is the number of bytes to read/write (must be less than the length of the buffer, also should be less than 64k since msg.len is u16.). Returned is the actual number of bytes read/written.

To communicate with the Nunchuk, you must send a handshake signal. In the probe() function, you will send "0xf0, 0x55" to initialize the first register of the Nunchuk and "0xFB, 0x00" to initialize the second register. Then, send one byte "0x00" each time you request data from the Nunchuck. You can see below the nunchuk_read_registers() function, which will read the data from the Nunchuk in 6 byte chunks:

```
static int nunchuk_read_registers(struct i2c_client *client, u8 *buf, int buf_size)
{
    int status;

    mdelay(10);

    buf[0] = 0x00;
    status = i2c_master_send(client, buf, 1);
    if (status >= 0 && status != 1)
        return -EIO;
    if (status < 0)
        return status;

    mdelay(10);

    status = i2c_master_recv(client, buf, buf_size);
    if (status >= 0 && status != buf_size)
        return -EIO;
    if (status < 0)
        return status;

    return 0;
}
```

The following image shows the data stream (six bytes) coming from the Nunchuck controller. First 2 bytes are the X and Y axis data of the Joystick. Next 3 bytes are the X, Y and Z axis data of the accelerometer sensor. The last byte includes the 2 lower bits of the accelerometer axes and the c-button and z-button status.

Data byte receive							Address
Joystick X							0x00
Joystick Y							0x01
Accelerometer X (bit 9 to bit 2 for 10-bit resolution)							0x02
Accelerometer Y (bit 9 to bit 2 for 10-bit resolution)							0x03
Accelerometer Z (bit 9 to bit 2 for 10-bit resolution)							0x04
Accel. Z bit 1	Accel. Z bit 0	Accel. Y bit 1	Accel. Y bit 0	Accel. X bit 1	Accel. X bit 0	C-button	Z-button
							0x05

Byte 0x00 : X-axis data of the joystick

Byte 0x01 : Y-axis data of the joystick

Byte 0x02 : X-axis data of the accellerometer sensor

Byte 0x03 : Y-axis data of the accellerometer sensor

Byte 0x04 : Z-axis data of the accellerometer sensor

Byte 0x05 : bit 0 as Z button status - 0 = pressed and 1 = release

 bit 1 as C button status - 0 = pressed and 1 = release

 bit 2 and 3 as 2 lower bit of X-axis data of the accellerometer sensor

 bit 4 and 5 as 2 lower bit of Y-axis data of the accellerometer sensor

 bit 6 and 7 as 2 lower bit of Z-axis data of the accellerometer sensor

Input framework driver setup

These are the main code sections:

1. Include the required header files:

```
#include <linux/input.h>
#include linux/input-polldev.h /* struct input_polled_dev, input_allocate_polled_
device(), input_register_polled_device() */
```
2. The device model needs to keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices (devices handled by subsystems, like the Input subsystem in this case). This need is typically implemented by creating a private data structure to manage the device and implement such pointers between the physical and logical worlds. Add the following private structure definition to your driver code:

```

struct nunchuk_dev {
    struct input_polled_dev *polled_input;
    struct i2c_client *client;
};

```

3. In the `nunchuk_probe()` function, declare an instance of the previous structure and allocate it:

```

struct nunchuk_dev *nunchuk;
nunchuk = devm_kzalloc(&client->dev, sizeof(*nunchuk), GFP_KERNEL);

```

4. To be able to access your private data structure in other functions of the driver, you need to attach it to the `i2c_client` structure using the `i2c_set_clientdata()` function. This function stores `nunchuk` in `client->dev->driver_data`. You can retrieve the `nunchuk` pointer to the private structure by using the `i2c_get_clientdata(client)` function:

```

i2c_set_clientdata(client, nunchuk); /* Write it in the probe() function */
nunchuk = i2c_get_clientdata(client); /* Write it in the remove() function */

```

5. Allocate the `input_polled_dev` structure in `probe()`:

```

polled_device = devm_input_allocate_polled_device(&client->dev);

```

6. Initialize the polled input device. Keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices:

```

nunchuk->client = client; /* Store a pointer to the I2C device in the global structure,
needed for exchanging data with the nunchuk device */

polled_device->private = nunchuk; /* struct polled_device can store the driver-specific
data in void *private. Place the pointer to the private structure here; in this way, you
will be able to recover the nunchuk pointer later (for example, in the nunchuk_poll()
function) */

polled_device->poll_interval = 50; /* Callback interval */

polled_device->poll = nunchuk_poll; /* Callback that will be called every 50 ms interval
*/

polled_device->input->dev.parent = &client->dev; /* Keep pointers between physical
devices and logical devices */

polled_device->input->name = "WII Nunchuk"; /* Input sub-device parameters that will
appear in log on registering the device */

polled_device->input->id.bustype = BUS_I2C; /* Input sub-device parameters */

```

7. Set event types and event codes for the Nunchuk device:

```

/* Set EV_KEY type events and from those BTN_C and BTN_Z event codes */
set_bit(EV_KEY, input->evbit);
set_bit(BTN_C, input->keybit); /* buttons */
set_bit(BTN_Z, input->keybit);

/*
 * Set EV_ABS type events and from those

```



```

    * ABS_X, ABS_Y, ABS_RX, ABS_RY and ABS_RZ event codes
    */
    set_bit(EV_ABS, input->evbit);
    set_bit(ABS_X, input->absbit); /* joystick */
    set_bit(ABS_Y, input->absbit);
    set_bit(ABS_RX, input->absbit); /* accelerometer */
    set_bit(ABS_RY, input->absbit);
    set_bit(ABS_RZ, input->absbit);

    /*
     * Fill additional fields in the input_dev struct for
     * each absolute axis nunchuk has
     */
    input_set_abs_params(input, ABS_X, 0x00, 0xff, 0, 0);
    input_set_abs_params(input, ABS_Y, 0x00, 0xff, 0, 0);

    input_set_abs_params(input, ABS_RX, 0x00, 0x3ff, 0, 0);
    input_set_abs_params(input, ABS_RY, 0x00, 0x3ff, 0, 0);
    input_set_abs_params(input, ABS_RZ, 0x00, 0x3ff, 0, 0);

```

8. Register in probe() and unregister in remove() the polled_input device to the input core. Once registered, the device is global for the rest of the driver functions until it is unregistered. After this call, the device is ready to accept requests from user space applications.

```

    input_register_polled_device(nunchuk->polled_input);
    input_unregister_polled_device(nunchuk->polled_input);

```

9. Write the nunchuk_poll() function. This function will be called every 50 ms. Inside nunchuk_poll(), you will call nunchuk_read_registers(), which read data from the Nunchuk device. The first parameter of the nunchuk_read_registers() function is a pointer to the i2c_client structure. This pointer will allow you to get the Nunchuk I2C address (0x52). The client pointer will be retrieved from client->address using the following lines of code:

```

    nunchuk = polled_input->private;
    client = nunchuk->client;

```

The first thing you should do is place a 10 ms delay at the beginning of the nunchuk_read_registers() function using the mdelay() function. This delay will separate the following I2C action from any previous I2C action. If you look through the Nunchuk documentation, you will see that each time you want to read from the Nunchuk device, you must first send the byte 0x00, then the Nunchuk will return 6 bytes of data. Therefore the next thing your nunchuk_read_registers() function should do is send the 0x00 byte by using the i2c_master_send() function. This action should be immediately followed by a 10 ms delay. Finally, nunchuk_read_registers() will read six bytes of data from the Nunchuk device and store them in buf using the i2c_master_recv() function.

You will store the buf[0] and buf[1] joystick values in the joy_x and joy_y variables. You will also get the C button and Z button status from the buf[5] variable and store it in the c_button

and `z_button` variables. The accelerometer data for its three axes will be retrieved from `buf[2]` and `buf[5]` and stored in the `accel_x`, `accel_y` and `accel_z` variables.

Finally, you will report the events to the Input subsystem. The `input_sync()` function will tell those who receive the events that a complete report has been sent.

```
static int nunchuk_read_registers(struct i2c_client *client, u8 *buf, int buf_size)
{
    mdelay(10);
    buf[0] = 0x00;
    i2c_master_send(client, buf, 1);
    mdelay(10);
    i2c_master_recv(client, buf, buf_size);
    return 0;
}

/*
 * Poll handler function reads the hardware,
 * queues events to be reported (input_report_*)
 * and flushes the queued events (input_sync)
 */
static void nunchuk_poll(struct input_polled_dev *polled_input)
{
    u8 buf[6];
    int joy_x, joy_y, z_button, c_button, accel_x, accel_y, accel_z;
    struct i2c_client *client;
    struct nunchuk_dev *nunchuk;

    /*
     * Recover the global nunchuk structure and from it the client address
     * to establish an I2C transaction with the nunchuck device
     */
    nunchuk = polled_input->private;
    client = nunchuk->client;

    /* Read the registers of the nunchuk device */
    nunchuk_read_registers(client, buf, ARRAY_SIZE(buf))

    joy_x = buf[0];
    joy_y = buf[1];

    /* Bit 0 indicates if Z button is pressed */
    z_button = (buf[5] & BIT(0)? 0 : 1);
    /* Bit 1 indicates if C button is pressed */
    c_button = (buf[5] & BIT(1)? 0 : 1);

    accel_x = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
    accel_y = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
    accel_z = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);

    /* Report events to the input system */
    input_report_abs(polled_input->input, ABS_X, joy_x);
    input_report_abs(polled_input->input, ABS_Y, joy_y);
```

```

    input_event(polled_input->input, EV_KEY, BTN_Z, z_button);
    input_event(polled_input->input, EV_KEY, BTN_C, c_button);

    input_report_abs(polled_input->input, ABS_RX, accel_x);
    input_report_abs(polled_input->input, ABS_RY, accel_y);
    input_report_abs(polled_input->input, ABS_RZ, accel_z);

    /*
     * Tell those who receive the events
     * that a complete report has been sent
     */
    input_sync(polled_input->input);
}

```

10. Create a new `linux_5.4_nunchuk_drivers` folder inside the `linux_5.4_rpi3_drivers` folder:

```
~/linux_5.4_rpi3_drivers$ mkdir linux_5.4_nunchuk_drivers
```

11. Create a new `nunchuk.c` file and a `Makefile` file in the `linux_5.4_nunchuk_drivers` folder, and add `nunchuk.o` to your `Makefile` `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_nunchuk_drivers$ make
~/linux_5.4_rpi3_drivers/linux_5.4_nunchuk_drivers$ make deploy
```

12. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

13. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 10-2: `nunchuk.c`

```

#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/input.h>
#include <linux/input-polldev.h>

/* Create private structure */
struct nunchuk_dev {
    struct input_polldev *polled_input;
    struct i2c_client *client;
};

static int nunchuk_read_registers(struct i2c_client *client, u8 *buf, int buf_size)
{
    int status;

```

```

    mdelay(10);

    buf[0] = 0x00;
    status = i2c_master_send(client, buf, 1);
    if (status >= 0 && status != 1)
        return -EIO;
    if (status < 0)
        return status;

    mdelay(10);

    status = i2c_master_recv(client, buf, buf_size);
    if (status >= 0 && status != buf_size)
        return -EIO;
    if (status < 0)
        return status;

    return 0;
}

/*
 * Poll handler function reads the hardware,
 * queues events to be reported (input_report_*)
 * and flushes the queued events (input_sync)
 */
static void nunchuk_poll(struct input_polled_dev *polled_input)
{
    u8 buf[6];
    int joy_x, joy_y, z_button, c_button, accel_x, accel_y, accel_z;
    struct i2c_client *client;
    struct nunchuk_dev *nunchuk;

    /*
     * Recover the global nunchuk structure and from it the client address
     * to establish an I2C transaction with the nunchuk device
     */
    nunchuk = polled_input->private;
    client = nunchuk->client;

    /* Read the registers of the nunchuk device */
    if (nunchuk_read_registers(client, buf, ARRAY_SIZE(buf)) < 0)
    {
        dev_info(&client->dev, "Error reading the nunchuk registers.\n");
        return;
    }

    joy_x = buf[0];
    joy_y = buf[1];

    /* Bit 0 indicates if Z button is pressed */
    z_button = (buf[5] & BIT(0)) ? 0 : 1;
    /* Bit 1 indicates if C button is pressed */
    c_button = (buf[5] & BIT(1)) ? 0 : 1;

```

```

accel_x = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
accel_y = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
accel_z = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);

/* Report events to the input system */
input_report_abs(polled_input->input, ABS_X, joy_x);
input_report_abs(polled_input->input, ABS_Y, joy_y);

input_event(polled_input->input, EV_KEY, BTN_Z, z_button);
input_event(polled_input->input, EV_KEY, BTN_C, c_button);

input_report_abs(polled_input->input, ABS_RX, accel_x);
input_report_abs(polled_input->input, ABS_RY, accel_y);
input_report_abs(polled_input->input, ABS_RZ, accel_z);

/*
 * Tell those who receive the events
 * that a complete report has been sent
 */
input_sync(polled_input->input);
}

static int nunchuk_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    int ret;
    u8 buf[2];
    struct device *dev = &client->dev;

    /* Declare a pointer to the private structure */
    struct nunchuk_dev *nunchuk;

    /* Declare pointers to input_dev and input_polled_dev structures */
    struct input_dev *input;
    /* For devices that can be polled on a timer basis */
    struct input_polled_dev *polled_device;

    dev_info(&client->dev, "nunchuck_probe() function is called.\n");

    /* Allocate private structure for new device */
    nunchuk = devm_kzalloc(&client->dev, sizeof(*nunchuk), GFP_KERNEL);
    if (nunchuk == NULL)
        return -ENOMEM;

    /* Associate client->dev with nunchuk private structure */
    i2c_set_clientdata(client, nunchuk);

    /* Allocate the struct input_polled_dev */
    polled_device = devm_input_allocate_polled_device(&client->dev);
    if (!polled_device) {
        dev_err(dev, "unable to allocate input device\n");
        return -ENOMEM;
    }

    /* Store the client device in the global structure */

```

```

nunchuk->client = client;

/* Initialize the polled input device */
/* To recover nunchuk in the poll() function */
polled_device->private = nunchuk;

/* Fill in the poll interval */
polled_device->poll_interval = 50;

/* Fill in the poll handler */
polled_device->poll = nunchuk_poll;

polled_device->input->dev.parent = &client->dev;

polled_device->input->name = "WII Nunchuk";
polled_device->input->id.bustype = BUS_I2C;

/*
 * Store the polled device in the global structure
 * to recover it in the remove() function
 */
nunchuk->polled_input = polled_device;

input = polled_device->input;

/* Set EV_KEY type events and from those BTN_C and BTN_Z event codes */
set_bit(EV_KEY, input->evbit);
set_bit(BTN_C, input->keybit); /* buttons */
set_bit(BTN_Z, input->keybit);

/*
 * Set EV_ABS type events and from those
 * ABS_X, ABS_Y, ABS_RX, ABS_RY and ABS_RZ event codes
 */
set_bit(EV_ABS, input->evbit);
set_bit(ABS_X, input->absbit); /* joystick */
set_bit(ABS_Y, input->absbit);
set_bit(ABS_RX, input->absbit); /* accelerometer */
set_bit(ABS_RY, input->absbit);
set_bit(ABS_RZ, input->absbit);

/*
 * Fill additional fields in the input_dev struct for
 * each absolute axis nunchuk has
 */
input_set_abs_params(input, ABS_X, 0x00, 0xff, 0, 0);
input_set_abs_params(input, ABS_Y, 0x00, 0xff, 0, 0);

input_set_abs_params(input, ABS_RX, 0x00, 0x3ff, 0, 0);
input_set_abs_params(input, ABS_RY, 0x00, 0x3ff, 0, 0);
input_set_abs_params(input, ABS_RZ, 0x00, 0x3ff, 0, 0);

/* Nunchuk handshake */
buf[0] = 0xf0;

```

```

    buf[1] = 0x55;
    ret = i2c_master_send(client, buf, 2);
    if (ret >= 0 && ret != 2)
        return -EIO;
    if (ret < 0)
        return ret;

    udelay(1);

    buf[0] = 0xfb;
    buf[1] = 0x00;
    ret = i2c_master_send(client, buf, 1);
    if (ret >= 0 && ret != 1)
        return -EIO;
    if (ret < 0)
        return ret;

    /* Finally, register the input device */
    ret = input_register_polled_device(nunchuk->polled_input);
    if (ret < 0)
        return ret;

    return 0;
}

static int nunchuk_remove(struct i2c_client *client)
{
    struct nunchuk_dev *nunchuk;
    nunchuk = i2c_get_clientdata(client);
    input_unregister_polled_device(nunchuk->polled_input);
    dev_info(&client->dev, "nunchuk_remove()\n");

    return 0;
}

/* Add entries to Device Tree */
static const struct of_device_id nunchuk_of_match[] = {
    { .compatible = "nunchuk"},
    {}
};
MODULE_DEVICE_TABLE(of, nunchuk_of_match);

static const struct i2c_device_id nunchuk_id[] = {
    { "nunchuk", 0 },
    {}
};
MODULE_DEVICE_TABLE(i2c, nunchuk_id);

/* Create struct i2c_driver */
static struct i2c_driver nunchuk_driver = {
    .driver = {
        .name = "nunchuk",
        .owner = THIS_MODULE,
        .of_match_table = nunchuk_of_match,
    },
};

```

```

    },
    .probe = nunchuk_probe,
    .remove = nunchuk_remove,
    .id_table = nunchuk_id,
};

/* Register to I2C bus as a driver */
module_i2c_driver(nunchuk_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a Nunchuk Wii I2C driver");

```

nunchuk.ko demonstration

Load the nunchuk module:

```

root@raspberrypi:/home/pi# insmod nunchuk.ko
nunchuk: loading out-of-tree module taints kernel.
nunchuk 1-0052: nunchuk_probe() function is called.
input: WII Nunchuk as /devices/platform/soc/3f804000.i2c/i2c-1/1-0052/input/input0

```

Execute the evtest application and play with the nunchuk device:

```

root@raspberrypi:/home/pi# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      WII Nunchuk
Select the device event number [0-0]: 0
Input driver version is 1.0.1
Input device ID: bus 0x18 vendor 0x0 product 0x0 version 0x0
Input device name: "WII Nunchuk"
Supported events:
Event type 0 (EV_SYN)
Event type 1 (EV_KEY)
  Event code 306 (BTN_C)
  Event code 309 (BTN_Z)
Event type 3 (EV_ABS)
  Event code 0 (ABS_X)
    Value      126
    Min         0
    Max       255
  Event code 1 (ABS_Y)
    Value      130
    Min         0
    Max       255
  Event code 3 (ABS_RX)
    Value      669
    Min         0
    Max     1023
  Event code 4 (ABS_RY)
    Value      513

```



```

Min      0
Max      1023
Event code 5 (ABS_RZ)
Value    634
Min      0
Max      1023

```

Properties:

Testing ... (interrupt to exit)

```

Event: time 1608594499.723581, type 3 (EV_ABS), code 3 (ABS_RX), value 669
Event: time 1608594499.723581, type 3 (EV_ABS), code 4 (ABS_RY), value 513
Event: time 1608594499.723581, type 3 (EV_ABS), code 5 (ABS_RZ), value 634
Event: time 1608594499.723581, ----- SYN_REPORT -----
Event: time 1608594499.803433, type 3 (EV_ABS), code 3 (ABS_RX), value 580
Event: time 1608594499.803433, type 3 (EV_ABS), code 4 (ABS_RY), value 482
Event: time 1608594499.803433, type 3 (EV_ABS), code 5 (ABS_RZ), value 665
Event: time 1608594499.803433, ----- SYN_REPORT -----
Event: time 1608594499.883281, type 3 (EV_ABS), code 3 (ABS_RX), value 490
Event: time 1608594499.883281, type 3 (EV_ABS), code 4 (ABS_RY), value 451
Event: time 1608594499.883281, type 3 (EV_ABS), code 5 (ABS_RZ), value 698
Event: time 1608594499.883281, ----- SYN_REPORT -----
Event: time 1608594499.963330, type 3 (EV_ABS), code 3 (ABS_RX), value 401
Event: time 1608594499.963330, type 3 (EV_ABS), code 4 (ABS_RY), value 421
Event: time 1608594499.963330, type 3 (EV_ABS), code 5 (ABS_RZ), value 730
Event: time 1608594499.963330, ----- SYN_REPORT -----
Event: time 1608594500.043247, type 3 (EV_ABS), code 3 (ABS_RX), value 387
Event: time 1608594500.043247, type 3 (EV_ABS), code 4 (ABS_RY), value 426
Event: time 1608594500.043247, type 3 (EV_ABS), code 5 (ABS_RZ), value 726
Event: time 1608594500.043247, ----- SYN_REPORT -----
Event: time 1608594500.123308, type 1 (EV_KEY), code 309 (BTN_Z), value 1
Event: time 1608594500.123308, type 3 (EV_ABS), code 3 (ABS_RX), value 388
Event: time 1608594500.123308, type 3 (EV_ABS), code 4 (ABS_RY), value 430
Event: time 1608594500.123308, type 3 (EV_ABS), code 5 (ABS_RZ), value 728
Event: time 1608594500.123308, ----- SYN_REPORT -----
Event: time 1608594500.203264, type 3 (EV_ABS), code 3 (ABS_RX), value 387
Event: time 1608594500.203264, type 3 (EV_ABS), code 4 (ABS_RY), value 429
Event: time 1608594500.203264, type 3 (EV_ABS), code 5 (ABS_RZ), value 730
Event: time 1608594500.203264, ----- SYN_REPORT -----
Event: time 1608594500.283249, type 3 (EV_ABS), code 3 (ABS_RX), value 389
Event: time 1608594500.283249, type 3 (EV_ABS), code 4 (ABS_RY), value 434
Event: time 1608594500.283249, type 3 (EV_ABS), code 5 (ABS_RZ), value 733
Event: time 1608594500.283249, ----- SYN_REPORT -----

```

Exit with ^C:

```
root@raspberrypi:/home/pi#
```

Remove the nunchuk module:

```
root@raspberrypi:/home/pi# rmmod nunchuk.ko
```

LAB 10.3: Nunchuk applications

In the previous lab, you have used the `evtest` application to display all the event types and event codes (and their values) generated by the Nunchuk device. In this LAB 10.3, you will use the `python-evdev` package, which will allow you to create python applications that read and write input events on Linux. The `evdev` interface serves the purpose of passing events generated in the kernel directly to user space through character devices that are typically located in `/dev/input/`.

You will use `pip` to install `python-evdev` on the Raspberry Pi:

```
root@raspberrypi:/home/pi# pip3 install evdev
```

Create a new `python_apps` folder inside the `nunchuk_drivers` folder, where you will store the python applications developed in this LAB 10.3:

```
~/linux_5.4_rpi3_drivers/nunchuk_drivers$ mkdir python_apps
```

The applications developed in this LAB 10.3 will not receive the event codes `ABS_RX`, `ABS_RY` and `ABS_RZ` (generated by the Nunchuk accelerometer device), so you will comment the next lines of code in bold in the Nunchuk driver:

```
static int nunchuk_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    int ret;
    u8 buf[2];
    struct device *dev = &client->dev;

    [...]

    /* Set EV_KEY type events and from those BTN_C and BTN_Z event codes */
    set_bit(EV_KEY, input->evbit);
    set_bit(BTN_C, input->keybit); /* buttons */
    set_bit(BTN_Z, input->keybit);

    /*
     * Set EV_ABS type events and from those
     * ABS_X, ABS_Y, ABS_RX, ABS_RY and ABS_RZ event codes
     */
    set_bit(EV_ABS, input->evbit);
    set_bit(ABS_X, input->absbit); /* joystick */
    set_bit(ABS_Y, input->absbit);
    //set_bit(ABS_RX, input->absbit); /* accelerometer */
    //set_bit(ABS_RY, input->absbit);
    //set_bit(ABS_RZ, input->absbit);

    /*
     * Fill additional fields in the input_dev struct for
     * each absolute axis nunchuk has
     */
    input_set_abs_params(input, ABS_X, 0x00, 0xff, 0, 0);
    input_set_abs_params(input, ABS_Y, 0x00, 0xff, 0, 0);
```

```

/*input_set_abs_params(input, ABS_RX, 0x00, 0x3ff, 0, 0);
input_set_abs_params(input, ABS_RY, 0x00, 0x3ff, 0, 0);
input_set_abs_params(input, ABS_RZ, 0x00, 0x3ff, 0, 0);*/

[...]

/* Finally, register the input device */
ret = input_register_polled_device(nunchuk->polled_input);
if (ret < 0)
    return ret;

return 0;
}

static void nunchuk_poll(struct input_polled_dev *polled_input)
{
    u8 buf[6];
    int joy_x, joy_y, z_button, c_button, accel_x, accel_y, accel_z;
    struct i2c_client *client;
    struct nunchuk_dev *nunchuk;

    [...]

    /* Bit 0 indicates if Z button is pressed */
    z_button = (buf[5] & BIT(0)? 0 : 1);
    /* Bit 1 indicates if C button is pressed */
    c_button = (buf[5] & BIT(1)? 0 : 1);

    /*accel_x = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
    accel_y = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
    accel_z = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);*/

    /* Report events to the input system */
    input_report_abs(polled_input->input, ABS_X, joy_x);
    input_report_abs(polled_input->input, ABS_Y, joy_y);

    input_event(polled_input->input, EV_KEY, BTN_Z, z_button);
    input_event(polled_input->input, EV_KEY, BTN_C, c_button);

    /*input_report_abs(polled_input->input, ABS_RX, accel_x);
    input_report_abs(polled_input->input, ABS_RY, accel_y);
    input_report_abs(polled_input->input, ABS_RZ, accel_z);*/

    /*
     * Tell those who receive the events
     * that a complete report has been sent
     */
    input_sync(polled_input->input);
}

```

Build and deploy the modified kernel module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/nunchuk_drivers$ make
~/linux_5.4_rpi3_drivers/nunchuk_drivers$ make deploy
```

The python-evdev package also comes with a small command-line program for listing and monitoring input devices. You can follow the next instructions to test the program:

Load the nunchuk module:

```
root@raspberrypi:/home/pi# insmod nunchuk.ko
nunchuk: loading out-of-tree module taints kernel.
nunchuk 1-0052: nunchuk_probe() function is called.
input: WII Nunchuk as /devices/platform/soc/3f804000.i2c/i2c-1/1-0052/input/input0
```

Execute the evtest application included in the python-evdev package:

```
root@raspberrypi:/home/pi# python3 -m evdev.evtest
```

ID	Device	Name	Phys
0	/dev/input/event0	WII Nunchuk	

```
Select devices [0-0]: 0
Listening for events (press ctrl-c to exit) ...
time 1612525084.315536 type 3 (EV_ABS), code 0 (ABS_X), value 147
time 1612525084.315536 ----- SYN_REPORT -----
time 1612525084.375653 type 3 (EV_ABS), code 0 (ABS_X), value 251
time 1612525084.375653 ----- SYN_REPORT -----
time 1612525084.435532 type 3 (EV_ABS), code 0 (ABS_X), value 255
time 1612525084.435532 ----- SYN_REPORT -----
time 1612525084.555543 type 3 (EV_ABS), code 0 (ABS_X), value 126
time 1612525084.555543 ----- SYN_REPORT -----
time 1612525087.015533 type 1 (EV_KEY), code 309 (BTN_Z), value 1
time 1612525087.015533 ----- SYN_REPORT -----
time 1612525087.915539 type 1 (EV_KEY), code 309 (BTN_Z), value 0
time 1612525087.915539 ----- SYN_REPORT -----
time 1612525090.135532 type 1 (EV_KEY), code 306 (BTN_C), value 1
time 1612525090.135532 ----- SYN_REPORT -----
time 1612525090.795537 type 1 (EV_KEY), code 306 (BTN_C), value 0
time 1612525090.795537 ----- SYN_REPORT -----
time 1612525095.295543 type 3 (EV_ABS), code 0 (ABS_X), value 255
time 1612525095.295543 ----- SYN_REPORT -----
time 1612525095.475548 type 3 (EV_ABS), code 0 (ABS_X), value 173
time 1612525095.475548 ----- SYN_REPORT -----
time 1612525095.535552 type 3 (EV_ABS), code 0 (ABS_X), value 126
time 1612525095.535552 ----- SYN_REPORT -----
time 1612525095.955535 type 3 (EV_ABS), code 1 (ABS_Y), value 26
time 1612525095.955535 ----- SYN_REPORT -----
time 1612525096.13554 type 3 (EV_ABS), code 0 (ABS_X), value 132
time 1612525096.13554 ----- SYN_REPORT -----
```

joystick_led.py application

You will develop now your first python application called `joystick_led`. This application will read an event type `EV_KEY` with event code `BTN_C` and an event type `EV_ABS` with event code `ABS_X`. When you press the C button on the Nunchuk device, it will be generated an event code `BTN_C` with value == 1, and an LED will be ON. When you release the C button on the Nunchuk device, it will be generated an event code `BTN_C` with value == 0, and the LED will be OFF. If you move the X axis of the Analog joystick, it will be generated an `ABS_X` event code with value == 126 if the joystick is centered, value > 126 if you are moving it to the right, and value < 126 if you are moving it to the left.

Connect any LED of the Color click eval board (<https://www.mikroe.com/color-click>) to the GPIO17 pin of the Raspberry Pi connector.

Create a `joystick_led.py` application, and store it in the `python_apps` folder which was created previously. Write the code of the following Listing 10.3 to the `joystick_led.py` file, and send it to the Raspberry Pi.

```
PC:~/linux_5.4_rpi3_drivers/nunchuk_drivers/python_apps$ scp joystick_led.py root@10.0.0.10:/home/pi/python-projects
```

Listing 10-3: joystick_led.py

```
from evdev import InputDevice, categorize, ecodes, KeyEvent
from gpiozero import LED
import sys

def main():
    joystick = InputDevice('/dev/input/event0')

    print(joystick)

    for event in joystick.read_loop():

        if event.type == ecodes.EV_KEY:
            keyevent = categorize(event)
            if keyevent.keycode == 'BTN_C':
                if keyevent.keystate == KeyEvent.key_down:
                    led.on()
                elif keyevent.keystate == KeyEvent.key_up:
                    led.off()

        elif event.type == ecodes.EV_ABS:
            absevent = categorize(event)
            if ecodes.bytype[absevent.event.type][absevent.event.code] == 'ABS_X':
                if absevent.event.value > 126:
                    print('right')
                print(absevent.event.value)
```

```

        elif absevent.event.value < 126:
            print('left')
            print(absevent.event.value)
        elif absevent.event.value == 126:
            print('centered')
            print(absevent.event.value)

if __name__ == '__main__':
    try:
        led = LED(17) // connect LED BLUE to the
        main()
    except (KeyboardInterrupt, EOFError):
        ret = 0
    led.close()
    sys.exit(ret)

```

joystick_led.py demonstration

Load the nunchuk module:

```

root@raspberrypi:/home/pi# insmod nunchuk.ko
nunchuk: loading out-of-tree module taints kernel.
nunchuk 1-0052: nunchuck_probe() function is called.
input: WII Nunchuk as /devices/platform/soc/3f804000.i2c/i2c-1/1-
0052/input/input0

```

Execute the joystick_led.py application. Press and release the C button of the nunchuk device to switch ON and OFF the LED. Move the x-axis of the analog joystick, and see the values on the command line:

```

root@raspberrypi:/home/pi/python-projects# python3 joystick_led.py
device /dev/input/event0, name "WII Nunchuk", phys ""
right
175
right
255
right
216
centered
126
left
55
left
54
left
0
centered
126
right
133
right
174

```

```
right
255
right
230
centered
126
```

Exit with ^C:

```
root@raspberrypi:/home/pi/python-projects# cd ..
```

Remove the nunchuk module:

```
root@raspberrypi:/home/pi# rmmod nunchuk.ko
nunchuk 1-0052: nunchuk_remove()
```

joystick_pwm.py application

You will develop now your second python application called joystick_pwm. This application will read an event type EV_ABS with event code ABS_X. When you move the X axis of the Analog joystick, it will be generated an ABS_X event code. The value of this event code will be used to modify the duty cycle of a PWM signal, changing the brightness of an LED. The maximum brightness will be set with a duty cycle of 100 (X axis far right), and the minimum brightness will be set with a duty cycle of 0 (X axis far left).

Connect any LED of the Color click eval board (<https://www.mikroe.com/color-click>) to the GPIO18 pin of the Raspberry Pi connector.

Create a joystick_pwm.py application, and store it in the python_apps folder. Write the code of the following Listing 10.4 to the joystick_pwm.py file, and send it to the Raspberry Pi:

```
PC:~/linux_5.4_rpi3_drivers/nunchuk_drivers/python_apps$ scp joystick_pwm.py root@10.0.0.10:/home/pi/python-projects
```

Listing 10-4: joystick_pwm.py

```
from evdev import InputDevice, categorize, ecodes, KeyEvent
import RPi.GPIO as GPIO
import time
import sys
import math

def main():
    joystick = InputDevice('/dev/input/event0')

    print(joystick)

    for event in joystick.read_loop():
        if event.type == ecodes.EV_ABS:
            absvent = categorize(event)
```

```

        if ecodes.bytype[absevent.event.type][absevent.event.code] == 'ABS_X':
            duty = math.floor((absevent.event.value * 100)/255)
            print(duty)
            pwm.ChangeDutyCycle(duty)    # Change duty cycle
            time.sleep(0.01)             # Delay of 10ms

if __name__ == '__main__':
    try:
        led = 12                        # connect red LED to the GPIO18
        GPIO.setwarnings(False)         # disable warnings
        GPIO.setmode(GPIO.BOARD)       # set pin numbering system. Using board pin numbering
        GPIO.setup(led,GPIO.OUT)
        pwm = GPIO.PWM(led,1000)        # create PWM instance with frequency
        pwm.start(0)                    # started PWM at 0% duty cycle
        main()
    except (KeyboardInterrupt, EOFError):
        ret = 0
    pwm.stop()
    GPIO.cleanup()
    sys.exit(ret)

```

joystick_pwm.py demonstration

Load the nunchuk module:

```

root@raspberrypi:/home/pi# insmod nunchuk.ko
nunchuk: loading out-of-tree module taints kernel.
nunchuk 1-0052: nunchuk_probe() function is called.
input: WII Nunchuk as /devices/platform/soc/3f804000.i2c/i2c-1/1-
0052/input/input0

```

Execute the joystick_pwm.py application. Move the x-axis of the analog joystick, and see how the LED changes its brightness:

```

root@raspberrypi:/home/pi/python-projects# python3 joystick_pwm.py
device /dev/input/event0, name "WII Nunchuk", phys ""
28
27
49
100
49
0
49

```

Exit with ^C:

```

root@raspberrypi:/home/pi/python-projects# cd ..

```

Remove the nunchuk module:

```

root@raspberrypi:/home/pi# rmmod nunchuk.ko
nunchuk 1-0052: nunchuk_remove()

```


joystick_pygame.py application

In the last application of the LAB 10.3, you will develop a simple Pygame application. Pygame is a set of Python modules designed for writing video games (<https://www.pygame.org/docs/>). The application will draw the X-Y Joystick values and also the C and Z button values of the Nunchuk device. You will connect your screen to the Raspberry Pi's HDMI port to see the Pygame application. You will also see the values of the Nunchuk device on the command line on your host PC.

Create a joystick_pygame.py application, and store it in the python_apps folder. Write the code of the following Listing 10.5 to the joystick_pygame.py file, and send it to the Raspberry Pi:

```
PC:~/linux_5.4_rpi3_drivers/nunchuk_drivers/python_apps$ scp joystick_pygame.py
root@10.0.0.10:/home/pi/python-projects
```

Listing 10-5: joystick_pygame.py

```
# Used to manage how fast the screen updates.
clock = pygame.time.Clock()

# Initialize the joysticks.
pygame.joystick.init()

# Get ready to print.
textPrint = TextPrint()

# ----- Main Program Loop -----
while not done:
    #
    # EVENT PROCESSING STEP
    #
    # Possible joystick actions: JOYAXISMOTION, JOYBALLMOTION, JOYBUTTONDOWN,
    # JOYBUTTONUP, JOYHATMOTION
    for event in pygame.event.get(): # User did something.
        if event.type == pygame.QUIT: # If user clicked close.
            done = True # Flag that we are done so we exit this loop.
        elif event.type == pygame.JOYBUTTONDOWN:
            print("Joystick button pressed.")
        elif event.type == pygame.JOYBUTTONUP:
            print("Joystick button released.")

    #
    # DRAWING STEP
    #
    # First, clear the screen to white. Don't put other drawing commands
    # above this, or they will be erased with this command.
    screen.fill(WHITE)
    textPrint.reset()

    # Get count of joysticks.
```

```

joystick_count = pygame.joystick.get_count()

print("Number of joysticks: {}".format(joystick_count))
textPrint.tprint(screen, "Number of joysticks: {}".format(joystick_count))
textPrint.indent()

# For each joystick:
for i in range(joystick_count):
    joystick = pygame.joystick.Joystick(i)
    joystick.init()

    try:
        jid = joystick.get_instance_id()
    except AttributeError:
        # get_instance_id() is an SDL2 method
        jid = joystick.get_id()
    print("Joystick {}".format(jid))
    textPrint.tprint(screen, "Joystick {}".format(jid))
    textPrint.indent()

    # Get the name from the OS for the controller/joystick.
    name = joystick.get_name()
    print("Joystick name: {}".format(name))
    textPrint.tprint(screen, "Joystick name: {}".format(name))

    try:
        guid = joystick.get_guid()
    except AttributeError:
        # get_guid() is an SDL2 method
        pass
    else:
        textPrint.tprint(screen, "GUID: {}".format(guid))

    # Usually axis run in pairs, up/down for one, and left/right for
    # the other.
    axes = joystick.get_numaxes()
    print("Number of axes: {}".format(axes))
    textPrint.tprint(screen, "Number of axes: {}".format(axes))
    textPrint.indent()

    for i in range(axes):
        axis = joystick.get_axis(i)
        print("Axis {} value: {:>6.3f}".format(i, axis))
        textPrint.tprint(screen, "Axis {} value: {:>6.3f}".format(i, axis))
    textPrint.unindent()

    buttons = joystick.get_numbuttons()
    print("Number of buttons: {}".format(buttons))
    textPrint.tprint(screen, "Number of buttons: {}".format(buttons))
    textPrint.indent()

    for i in range(buttons):
        button = joystick.get_button(i)
        print("Button {:>2} value: {}".format(i, button))

```

```

        textPrint.tprint(screen,
                        "Button {:>2} value: {}".format(i, button))
    textPrint.unindent()

#
# ALL CODE TO DRAW SHOULD GO ABOVE THIS COMMENT
#

# Go ahead and update the screen with what we've drawn.
pygame.display.flip()

# Limit to 20 frames per second.
clock.tick(20)

# Close the window and quit.
# If you forget this line, the program will 'hang'
# on exit if running from IDLE.
pygame.quit()

```

joystick_pygame.py demonstration

Load the nunchuk module:

```

root@raspberrypi:/home/pi# insmod nunchuk.ko
nunchuk: loading out-of-tree module taints kernel.
nunchuk 1-0052: nunchuck_probe() function is called.
input: WII Nunchuk as /devices/platform/soc/3f804000.i2c/i2c-1/1-
0052/input/input0

```

Execute the joystick_pygame.py application. Move the x-axis and the y-axis of the analog joystick, and press the C and Z buttons on the nunchuk device. You will see the values on the command line and also drawn on your screen:

```

root@raspberrypi:/home/pi/python-projects# python3 joystick_pygame.py
Number of joysticks: 1
Joystick 0
Joystick name: WII Nunchuk
Number of axes: 2
Axis 0 value: 0.000
Axis 1 value: 0.000
Number of buttons: 2
Button 0 value: 0
Button 1 value: 0
Joystick button released.
Joystick button released.
Number of joysticks: 1
Joystick 0
Joystick name: WII Nunchuk
Number of axes: 2
Axis 0 value: -0.008
Axis 1 value: 0.024
Number of buttons: 2
Button 0 value: 0

```

```
Button 1 value: 0
Number of joysticks: 1
Joystick 0
Joystick name: Wii Nunchuk
Number of axes: 2
Axis 0 value: -0.008
Axis 1 value: 0.024
Number of buttons: 2
Button 0 value: 0
Button 1 value: 0
Number of joysticks: 1
Joystick 0
Joystick name: Wii Nunchuk
Number of axes: 2
Axis 0 value: -0.008
Axis 1 value: 0.024
Number of buttons: 2
Button 0 value: 0
Button 1 value: 0
Joystick button pressed.
Number of joysticks: 1
Joystick 0
Joystick name: Wii Nunchuk
Number of axes: 2
Axis 0 value: -0.008
Axis 1 value: 0.024
Number of buttons: 2
Button 0 value: 1
Button 1 value: 0
Number of joysticks: 1
Joystick 0
Joystick name: Wii Nunchuk
Number of axes: 2
Axis 0 value: -0.008
Axis 1 value: 0.024
Number of buttons: 2
Button 0 value: 1
Button 1 value: 0
Number of joysticks: 1
Joystick 0
```

Exit with ^C:

```
^CTraceback (most recent call last):
  File "joystick_pygame.py", line 98, in <module>
```

```
    name = joystick.get_name()
KeyboardInterrupt
```

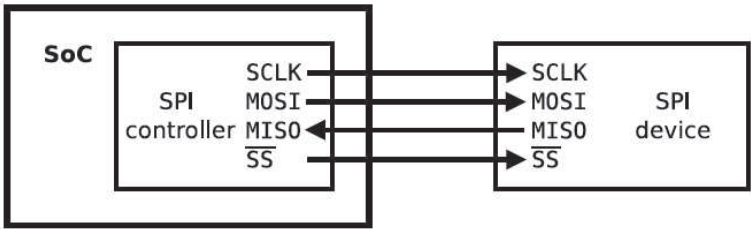
```
root@raspberrypi:/home/pi/python-projects# cd ..
```

Remove the nunchuk module:

```
root@raspberrypi:/home/pi# rmmmod nunchuk.ko
nunchuk 1-0052: nunchuk_remove()
```

Using SPI with Linux

The Serial Peripheral Interface (SPI) is a synchronous four wire serial link used to connect microprocessors to sensors, memories and peripherals. It's a simple "de facto" standard, not complicated enough to require a standardization body. The SPI uses a master/slave configuration.



The three signal wires hold a clock (SCK, often in the range of 1-20 MHz) and parallel data lines with "Master Out, Slave In" (MOSI) or "Master In, Slave Out" (MISO) signals. There are four clocking modes through which data is exchanged; mode-0 and mode-3 are most commonly used. Each clock cycle shifts data out and data in; the clock doesn't cycle except when there is a data bit to shift. Not all data bits are used though; not every protocol uses those full duplex capabilities.

SPI masters use a "chip select" line to activate a given SPI slave device, so those three signal wires may be connected to several chips in parallel. All SPI slaves support chipselects; they are usually active low signals, labeled nCSx for slave 'x' (e.g., nCS0). Some devices have other signals, often including an interrupt to the master.

The programming interface is structured around two kinds of drivers: the controller and protocol drivers. The **controller drivers** support the SPI master controller and drive hardware to control the clock and chip selects, shift data bits on/off wire, and configure basic SPI characteristics like clock frequency and mode. The SPI controller driver for the BCM2835 SoC is located at drivers/spi/spi-bcm2835aux.c in the kernel source tree. The **protocol drivers** support the SPI slave specific functionality, are based on messages, and transfer and rely on the controller driver to program SPI master hardware.

The I/O model is a set of queued messages. A single message (fundamental argument to all SPI subsystem read/write APIs) is an atomic sequence of transfers built from one or more spi_transfer objects, each of which wraps a full duplex SPI transfer which is processed and completed synchronously or asynchronously. When using synchronous request, the caller is blocked until the call succeeds. When using asynchronous request, you are periodically checking if the transaction is finished. The SPI controller driver manages access to those devices through a queue of spi_message

transactions, copying data between CPU memory and an SPI slave device. For each such message it queues, it calls the message's completion function when the transaction completes.

See the `at25_ee_read()` function (located in `drivers/misc/eeprom/at25.c`) as an example of an SPI transaction:

```
struct spi_transfer t[2];
struct spi_message m;
spi_message_init(&m);
memset(t, 0, sizeof t);

t[0].tx_buf = command;
t[0].len = at25->addrlen + 1;
spi_message_add_tail(&t[0], &m);

t[1].rx_buf = buf;
t[1].len = count;
spi_message_add_tail(&t[1], &m);
status = spi_sync(at25->spi, &m);
```

The basic I/O primitive is `spi_async()`. Async requests may be issued in any context (irq handler, task, etc.), and completion is reported by using a callback provided with the message. After any detected error, the chip is deselected and processing of that `spi_message` is aborted.

There are also synchronous wrappers like `spi_sync()`, and wrappers like `spi_read()`, `spi_write()` and `spi_write_then_read()`. These may be issued only in contexts that may sleep, and they're all clean layers over `spi_async()`.

The `spi_write_then_read()` call, and convenience wrappers around it, should only be used with small amounts of data where the cost of an extra copy may be ignored; for example, you can see below the `spi_w8r16()` wrapper, which writes an eight bit command and reads a sixteen bit response.

```
static inline ssize_t spi_w8r16(struct spi_device *spi, u8 cmd)
{
    ssize_t status;
    u16 result;

    status = spi_write_then_read(spi, &cmd, 1, &result, 2);

    /* return negative errno or unsigned value */
    return (status < 0) ? status : result;
}
```

The Linux SPI subsystem

The Linux **SPI subsystem** is based on the Linux device model and is composed of several drivers:

1. The **SPI bus core** of the SPI subsystem is located in the `spi.c` file under `drivers/spi/` directory. It is a collection of code that provides interface support between an SPI client driver and an SPI bus master. The SPI bus core registers itself with the kernel by using the `bus_register()` function and also defines a `bus_type` structure called `spi_bus_type`.

```
struct bus_type spi_bus_type = {
    .name           = "spi",
    .dev_groups      = spi_dev_groups,
    .match           = spi_match_device,
    .uevent          = spi_uevent,
};
EXPORT_SYMBOL_GPL(spi_bus_type);
```

The SPI core API is a set of functions (`spi_write_then_read()`, `spi_sync()`, `spi_async()`, etc.) used for an **SPI client driver** to manage SPI transactions with a device connected to an SPI bus.

2. The **SPI controller drivers** are located under `drivers/spi/` directory in the kernel source tree. The SPI controller is a platform device (declared in the Device Tree) that must be registered as a device to the platform bus via the `of_platform_populate()` function and registered with the SPI bus core as a driver by using the `module_platform_driver()` function.

```
static struct platform_driver bcm2835_spi_driver = {
    .driver = {
        .name = DRV_NAME,
        .of_match_table = bcm2835_spi_match,
    },
    .probe = bcm2835_spi_probe,
    .remove = bcm2835_spi_remove,
};
module_platform_driver(bcm2835_spi_driver);
```

The SPI controller driver includes a set of custom functions that issues read/writes to each SPI controller hardware I/O addresses. The SPI controller driver provides an `spi_master` structure per each probed SPI controller, then initializes the fields of `struct spi_controller` with the methods that interact with the SPI core and the SPI protocol (slave) drivers. These are some of the SPI master methods included in the `struct spi_controller` (declared in `include/linux/spi/spi.h`):

- `master->setup(struct spi_device *spi)`: This method sets up the device clock rate, SPI mode, and word sizes.
- `master->transfer_one(struct spi_master *master, struct spi_device *spi, struct spi_transfer *transfer)`: This method executes a single transfer while queuing transfers that

arrive in the meantime. When the transfer has finished, the `spi_finalize_current_transfer()` function is called so that the subsystem can issue the next transfer.

The `spi_register_controller()` function registers each SPI controller to the SPI bus core, publishing it to the rest of the system. At that time, the device nodes for the controller, and any pre-declared SPI devices will be made available; the driver model core will take care of binding them to the drivers. The SPI master driver needs to implement a mechanism to send the data on the SPI bus using the SPI device specified settings. It's the SPI master driver's responsibilities to operate the hardware to send out the data. Normally, the SPI master needs to implement:

- **A message queue:** To hold the messages from the SPI device driver.
- **A workqueue and workqueue thread:** To pump the messages from the message queue and start transfer.
- **A tasklet and tasklet handler:** To send the data on the hardware.
- **An interrupt handler:** To handle the interrupts during the transfer.

In the `probe()` function of the `drivers/spi/spi-bcm2835.c` driver, you can see the initialization and registration of an SPI master controller:

```
static int bcm2835_spi_probe(struct platform_device *pdev)
{
    struct spi_controller *ctlr;
    struct bcm2835_spi *bs;
    int err;

    ctlr = devm_spi_alloc_master(&pdev->dev, ALIGN(sizeof(*bs),
                                                dma_get_cache_alignment()));
    if (!ctlr)
        return -ENOMEM;

    platform_set_drvdata(pdev, ctlr);

    ctlr->use_gpio_descriptors = true;
    ctlr->mode_bits = BCM2835_SPI_MODE_BITS;
    ctlr->bits_per_word_mask = SPI_BPW_MASK(8);
    ctlr->num_chipselect = BCM2835_SPI_NUM_CS;
    ctlr->setup = bcm2835_spi_setup;
    ctlr->transfer_one = bcm2835_spi_transfer_one;
    ctlr->handle_err = bcm2835_spi_handle_err;
    ctlr->prepare_message = bcm2835_spi_prepare_message;
    ctlr->dev.of_node = pdev->dev.of_node;

    bs = spi_controller_get_devdata(ctlr);

    bs->regs = devm_platform_ioremap_resource(pdev, 0);
    if (IS_ERR(bs->regs))
        return PTR_ERR(bs->regs);
```



```

bs->clk = devm_clk_get(&pdev->dev, NULL);
if (IS_ERR(bs->clk)) {
    err = PTR_ERR(bs->clk);
    dev_err(&pdev->dev, "could not get clk: %d\n", err);
    return err;
}

bs->irq = platform_get_irq(pdev, 0);
if (bs->irq <= 0)
    return bs->irq ? bs->irq : -ENODEV;

clk_prepare_enable(bs->clk);

bcm2835_dma_init(ctlr, &pdev->dev, bs);

/* initialise the hardware with the default polarities */
bcm2835_wr(bs, BCM2835_SPI_CS,
          BCM2835_SPI_CS_CLEAR_RX | BCM2835_SPI_CS_CLEAR_TX);

err = devm_request_irq(&pdev->dev, bs->irq, bcm2835_spi_interrupt, 0,
                      dev_name(&pdev->dev), ctlr);
if (err) {
    dev_err(&pdev->dev, "could not request IRQ: %d\n", err);
    goto out_dma_release;
}

err = spi_register_controller(ctlr);
if (err) {
    dev_err(&pdev->dev, "could not register SPI controller: %d\n",
            err);
    goto out_dma_release;
}

bcm2835_debugfs_create(bs, dev_name(&pdev->dev));

return 0;

out_dma_release:
bcm2835_dma_release(ctlr, bs);
clk_disable_unprepare(bs->clk);
return err;
}

```

3. The **SPI protocol drivers** are located throughout `linux/drivers/`, depending on the type of device (for example, `linux/drivers/input/` for input devices). The driver code is specific to the device (accelerometer, digital analog converter, etc.) and uses the SPI core API to communicate with the SPI master driver and send/receive data to/from the SPI device.

For example, if the SPI client driver calls `spi_write_then_read()`, then this function calls `spi_sync()`, which in turn calls `__spi_sync()`. The `__spi_sync()` function calls `__spi_pump_messages()`, which processes the spi message queue and checks if there is any spi message

in the queue that needs processing, and if so calls out to the driver to initialize the hardware and transfer each message. The `__spi_pump_messages()` function is called both, from the kthread itself and also from inside `spi_sync()`; the queue extraction, handled at the top of the function should deal with this safely. Finally, `__spi_pump_messages()` calls the `transfer_one_message` method (initialized to the `bcm2835_spi_transfer_one()` function for the BCM2837 SoC).

Writing SPI client drivers

In this and in successive chapters, you will develop several SPI client drivers to control an accelerometer and a Mixed-Signal I/O Device. In the following sections, it will be covered the main steps to set up an SPI client driver.

SPI client driver registration

The SPI subsystem defines an `spi_driver` structure that is inherited from the `device_driver` structure and which must be instantiated and registered to the SPI bus core by each SPI client driver. Usually, you will implement a single driver structure and instantiate all clients from it. Remember, a driver structure contains general access routines and should be zero-initialized, except for fields with data you provide. See below a struct `spi_driver` definition for an SPI accelerometer device:

```
static struct spi_driver adxl345_driver = {
    .driver = {
        .name = "adxl345",
        .owner = THIS_MODULE,
        .of_match_table = adxl345_dt_ids,
    },
    .probe = adxl345_spi_probe,
    .remove = adxl345_spi_remove,
    .id_table = adxl345_id,
};
module_spi_driver(adxl345_driver);
```

The `module_spi_driver()` macro is used to register/unregister the driver.

In your Linux SPI driver, you will create an array of structures of type `of_device_id` where you specify compatible strings that hold the same value of the DT device node's compatible property. The `of_device_id` structure is declared as follows:

```
struct of_device_id {
    char name[32];
    char type[32];
    char compatible[128];
};
```

The `of_match_table` field (included in the `spi_driver` structure) is a pointer to an array of `of_device_id` structures that store the compatible strings supported by the driver:

```
static const struct of_device_id adxl345_dt_ids[] = {
    { .compatible = "arrow,adxl345", },
    { }
};
MODULE_DEVICE_TABLE(of, adxl345_dt_ids);
```

The driver's `probe()` function is called when the compatible field in one of the `of_device_id` entries matches with the compatible property of a DT device node. The `probe()` function is responsible to initialize the device with the configuration values obtained from the matched DT device node and also to register the device to the appropriate kernel framework.

In your SPI device driver, you will also define an array of `spi_device_id` structures:

```
static const struct spi_device_id adxl345_id[] = {
    { .name = "adxl345", },
    { }
};
MODULE_DEVICE_TABLE(spi, adxl345_id);
```

Declaration of SPI devices in the Device Tree

In the Device Tree, the SPI controller is typically declared in the `.dtsi` file that describes the processor. The SPI controller is normally declared with `status = "disabled"`. See below the declaration of the BCM2837 SPI controller (located in `arch/arm/boot/dts/bcm283x.dtsi`):

```
spi: spi@7e204000 {
    compatible = "brcm,bcm2835-spi";
    reg = <0x7e204000 0x200>;
    interrupts = <2 22>;
    clocks = <&clocks BCM2835_CLOCK_VPU>;
    #address-cells = <1>;
    #size-cells = <0>;
    status = "disabled";
};
```

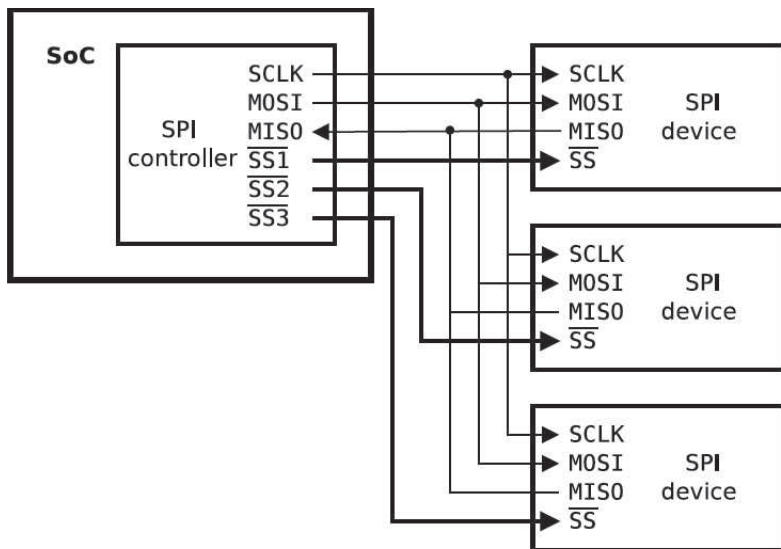
These are the required properties:

- `compatible`: "brcm,bcm2835-spi" will match with the compatible value of the `spi-bcm2835.c` driver.
- `reg`: Offset and length of the register set for the device.
- `interrupts`: List of interrupt specifiers, one for each interrupt output signal on the device.
- `clocks`: List of phandle and clock specifier pairs, one pair for each clock input to the device. See the clock bindings at [Documentation/devicetree/bindings/clock/clock-bindings.txt](#).
- `cs-gpios`: Specifies the gpio pins to be used for chip selects.

For the BCM2837 SoC, the `cs-gpios` property (located in `arch/arm/boot/dts/bcm2710-rpi-3-b.dts`) enables two chip selects:

```
cs-gpios = <gpio 8 1>, <gpio 7 1>;
```

You can see in the following figure an SPI controller with multiple chip selects connected to several SPI devices:



The Device Tree declaration of the SPI devices is done as sub-nodes of the SPI master controller at the board/platform level (`arch/arm/boot/dts/bcm2710-rpi-3-b.dts`). These are the required and optional properties:

- `reg`: (required) chip select address of device.
- `compatible`: (required) name of SPI device that should match with one of the driver's of `_device_id` compatible strings.
- `spi-max-frequency`: (required) maximum SPI clocking speed of device in Hz.
- `spi-cpol`: (optional) empty property indicating device requires inverse clock polarity (CPOL) mode.
- `spi-cpha`: (optional) empty property indicating device requires shifted clock phase (CPHA) mode.
- `spi-cs-high`: (optional) empty property indicating device requires chip select active high.
- `spi-3wire`: (optional) empty property indicating device requires 3-wire mode.
- `spi-lsb-first`: (optional) empty property indicating device requires LSB first mode.

- `spi-tx-bus-width`: (optional) the bus width (number of data wires) that is used for MOSI; defaults to 1 if not present.
- `spi-rx-bus-width`: (optional) the bus width (number of data wires) that is used for MISO; defaults to 1 if not present.
- `spi-rx-delay-us`: (optional) microsecond delay after a read transfer.
- `spi-tx-delay-us`: (optional) microsecond delay after a write transfer.

LAB 10.4: "SPI accel input device" module

Throughout the upcoming lab, you will implement your first driver for an SPI device. The driver will manage an accelerometer device connected to the SPI bus. You will use the same ADXL345 Accel click mikroBUS™ accessory board from the previous LAB 10.1.

To develop the new driver, you will draw on the mainlined ADXL345 Input 3-Axis Digital Accelerometer Linux Driver from Michael Hennerich, removing some features to simplify it for educational purposes. Your ADXL345 driver will only support SPI. See a description of the Michael Hennrich driver at <https://wiki.analog.com/resources/tools-software/linux-drivers/input-misc/adxl345>.

The driver will detect single tap motion on any of the 3 axis. The tap detection threshold is defined by the `THRESH_TAP` register (Address 0x1D). The `SINGLE_TAP` bit of the `INT_SOURCE` register (Address 0x30) is set when a single acceleration event greater than the value in the `THRESH_TAP` register (Address 0x1D) occurs in less time than is specified in the `DUR` register (Address 0x21). The single tap interrupt is triggered when the acceleration goes below the threshold, as long as `DUR` has not been exceeded (you can see page 28 of the ADXL345 data-sheet). You will select by default the tap motion detection only in the Z axis, enabling it by writing in the `TAP_AXES` (Address 0x2A) register.

LAB 10.4 hardware description

See below a description of the connections:

- Connect Raspberry Pi **GPIO8 (CE0)** to ADXL345 **CS (CS)**.
- Connect Raspberry Pi **SCLK** to ADXL345 **SCL (SCK)**
- Connect Raspberry Pi **MISO** to ADXL345 **SDO (MISO)**
- Connect Raspberry Pi **MOSI** to ADXL345 **SDI (MOSI)**
- Connect Raspberry Pi **GPIO23** to ADXL345 **INT1 (INT)**

LAB 10.4 Device Tree description

Open and modify the `bcm2710-rpi-3-b.dts` Device Tree file by adding the `adxl345@0` sub-node in the `spi0` controller master node. The `pinctrl-0` property of the `adxl345` node points to the `accel_int_pin` pin configuration node, where the `GPIO23` pad is multiplexed as a GPIO signal. The `int-gpios` property will make the `GPIO23` available to the driver so that you can set the pin direction to input and get the Linux IRQ number associated with this pin. The `reg` property provides the CS number. There are two chip selects inside the `spi0` node, but you will only use the first one `<gpio 8 1>` for the `ADXL345` device.

```
&spi0 {
    pinctrl-names = "default";
    pinctrl-0 = <&spi0_pins &spi0_cs_pins>;
    cs-gpios = <&gpio 8 1>, <&gpio 7 1>;

    Accel: ADXL345@0 {
        compatible = "arrow,adxl345";
        spi-max-frequency = <5000000>;
        spi-cpol;
        spi-cpha;
        reg = <0>;
        pinctrl-0 = <&accel_int_pin>;
        int-gpios = <&gpio 23 0>;
        interrupts = <23 1>;
        interrupt-parent = <&gpio>;
    };
};
```

The `GPIO23` pin is multiplexed as a GPIO signal in the `accel_int_pin` pin configuration node:

```
accel_int_pin: accel_int_pin {
    brcm,pins = <23>;
    brcm,function = <0>;    /* Input */
    brcm,pull = <0>;        /* none */
};
```

LAB 10.4 code description of the "SPI accel input device" module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/module.h>
#include <linux/input.h>
#include <linux/spi/spi.h>
#include <linux/of_gpio.h>
#include <linux/spi/spi.h>
#include <linux/interrupt.h>
```

2. Define the masks and macros which will generate the specific command byte of the SPI transaction (`spi_read()`, `spi_write()`, `spi_write_then_read()`, etc.):

```
#define ADXL345_CMD_MULTB          (1 << 6)
#define ADXL345_CMD_READ          (1 << 7)
#define ADXL345_WRITECMD(reg)     (reg & 0x3F)
#define ADXL345_READCMD(reg)      (ADXL345_CMD_READ | (reg & 0x3F))
#define ADXL345_READMB_CMD(reg)   (ADXL345_CMD_READ | ADXL345_CMD_MULTB \
                                   | (reg & 0x3F))
```

3. Define the registers of the ADXL345 device:

```
/* ADXL345 Register Map */
#define DEVID                      0x00 /* R Device ID */
#define THRESH_TAP                 0x1D /* R/W Tap threshold */
#define DUR                        0x21 /* R/W Tap duration */
#define TAP_AXES                   0x2A /* R/W Axis control for tap/double tap */
#define ACT_TAP_STATUS             0x2B /* R Source of tap/double tap */
#define BW_RATE                    0x2C /* R/W Data rate and power mode control */
#define POWER_CTL                  0x2D /* R/W Power saving features control */
#define INT_ENABLE                 0x2E /* R/W Interrupt enable control */
#define INT_MAP                    0x2F /* R/W Interrupt mapping control */
#define INT_SOURCE                 0x30 /* R Source of interrupts */
#define DATA_FORMAT               0x31 /* R/W Data format control */
#define DATA0                     0x32 /* R X-Axis Data 0 */
#define DATA1                     0x33 /* R X-Axis Data 1 */
#define DATAY0                     0x34 /* R Y-Axis Data 0 */
#define DATAY1                     0x35 /* R Y-Axis Data 1 */
#define DATAZ0                     0x36 /* R Z-Axis Data 0 */
#define DATAZ1                     0x37 /* R Z-Axis Data 1 */
#define FIFO_CTL                   0x38 /* R/W FIFO control */
```

4. Create the rest of `#define` to perform operations in the registers of the ADXL345 device and to pass some of them as arguments to several functions of the driver:

```
/* DEVIDs */
#define ID_ADXL345                 0xE5

/* INT_ENABLE/INT_MAP/INT_SOURCE Bits */
#define SINGLE_TAP                 (1 << 6)

/* TAP_AXES Bits */
#define TAP_X_EN                   (1 << 2)
#define TAP_Y_EN                   (1 << 1)
#define TAP_Z_EN                   (1 << 0)

/* BW_RATE Bits */
#define LOW_POWER                   (1 << 4)
#define RATE(x)                    ((x) & 0xF)

/* POWER_CTL Bits */
#define PCTL_MEASURE                (1 << 3)
#define PCTL_STANDBY               0X00

/* DATA_FORMAT Bits */
```

```

#define FULL_RES          (1 << 3)

/* FIFO_CTL Bits */
#define FIFO_MODE(x)      (((x) & 0x3) << 6)
#define FIFO_BYPASS       0
#define FIFO_FIFO         1
#define FIFO_STREAM       2
#define SAMPLES(x)        ((x) & 0x1F)

/* FIFO_STATUS Bits */
#define ADXL_X_AXIS       0
#define ADXL_Y_AXIS       1
#define ADXL_Z_AXIS       2

#define ADXL345_GPIO_NAME "int"

/* Macros to do SPI operations */
#define AC_READ(ac, reg)  ((ac)->bops->read((ac)->dev, reg))
#define AC_WRITE(ac, reg, val) ((ac)->bops->write((ac)->dev, reg, val))

```

5. Create the different structures of the driver:

```

/* Define a structure to hold SPI bus operations */
struct adxl345_bus_ops {
    u16 bustype;
    int (*read)(struct device *, unsigned char);
    int (*read_block)(struct device *, unsigned char, int, void *);
    int (*write)(struct device *, unsigned char, unsigned char);
};

struct axis_triple {
    int x;
    int y;
    int z;
};

/* Define a structure to hold specific driver's information */
struct adxl345_platform_data {
    u8 low_power_mode;
    u8 tap_threshold;
    u8 tap_duration;
    #define ADXL_TAP_X_EN          (1 << 2)
    #define ADXL_TAP_Y_EN          (1 << 1)
    #define ADXL_TAP_Z_EN          (1 << 0)
    u8 tap_axis_control;
    u8 data_rate;
    #define ADXL_FULL_RES          (1 << 3)
    #define ADXL_RANGE_PM_2g       0
    #define ADXL_RANGE_PM_4g       1
    #define ADXL_RANGE_PM_8g       2
    #define ADXL_RANGE_PM_16g      3
    u8 data_range;
    u32 ev_code_tap[3];
    u8 fifo_mode;
    u8 watermark;
};

```



```

/* Set initial adxl345 register values */
static const struct adxl345_platform_data adxl345_default_init = {
    .tap_threshold = 50,
    .tap_duration = 3,
    .tap_axis_control = ADXL_TAP_Z_EN,
    .data_rate = 8,
    .data_range = ADXL_FULL_RES,
    .fifo_mode = FIFO_BYPASS,
    .watermark = 0,
};

/* Define a private data structure */
struct adxl345 {
    struct gpio_desc *gpio;
    struct device *dev;
    struct input_dev *input;
    struct adxl345_platform_data pdata;
    struct axis_triple saved;
    u8 phys[32];
    int irq;
    u32 model;
    u32 int_mask;
    const struct adxl345_bus_ops *bops;
};

```

6. Initialize the `adxl345_bus_ops` structure with the functions that will perform the bus operations, and send it to the `adxl345_probe()` function as an argument:

```

static const struct adxl345_bus_ops adxl345_spi_bops = {
    .bustype      = BUS_SPI,
    .write        = adxl345_spi_write,
    .read         = adxl345_spi_read,
    .read_block   = adxl345_spi_read_block,
};

static int adxl345_spi_probe(struct spi_device *spi)
{
    /* Create a private structure */
    struct adxl345 *ac;

    /* initialize the driver and returns the initialized private struct */
    ac = adxl345_probe(&spi->dev, &adxl345_spi_bops);

    /* Attach the SPI device to the private structure */
    spi_set_drvdata(spi, ac);
    return 0;
}

```

7. See below an extract of the `adxl345_probe()` routine with the main lines of code commented:

```

struct adxl345 *adxl345_probe(struct device *dev, const struct adxl345_bus_ops *bops)
{
    /* declare your private structure */
    struct adxl345 *ac;

    /* Create the input device */

```

```

struct input_dev *input_dev;

/* Create pointer to const struct platform data */
const struct adxl345_platform_data *pdata;

/* Allocate private structure */
ac = devm_kzalloc(dev, sizeof(*ac), GFP_KERNEL);

/* Allocate the input_dev structure */
input_dev = devm_input_allocate_device(dev);

/*
 * Store the previously initialized platform data
 * in your private structure
 */
pdata = &adxl345_default_init; /* Points to const platform data */
ac->pdata = *pdata; /* Store values to pdata inside private ac */
pdata = &ac->pdata; /* change where pdata points, now to pdata in ac */

/* Store the input device in your private structure */
ac->input = input_dev;
ac->dev = dev; /* dev is &spi->dev */

/* Store the SPI operations in your private structure */
ac->bops = bops;

/* Initialize the input device */
input_dev->name = "ADXL345 accelerometer";
input_dev->phys = ac->phys;
input_dev->dev.parent = dev;
input_dev->id.product = ac->model;
input_dev->id.bustype = bops->bustype;

/* Attach the input device and the private structure */
input_set_drvdata(input_dev, ac);

/*
 * Set EV_KEY type event with 3 events code support.
 * The event is sent when a single tap interrupt is triggered
 */
__set_bit(EV_KEY, input_dev->evbit);
__set_bit(pdata->ev_code_tap[ADXL_X_AXIS], input_dev->keybit);
__set_bit(pdata->ev_code_tap[ADXL_Y_AXIS], input_dev->keybit);
__set_bit(pdata->ev_code_tap[ADXL_Z_AXIS], input_dev->keybit);

/*
 * Check if any of the axis has been enabled
 * and set the interrupt mask.
 * In this driver is only enabled the SINGLE_TAP interrupt
 */
if (pdata->tap_axis_control & (TAP_X_EN | TAP_Y_EN | TAP_Z_EN))
    ac->int_mask |= SINGLE_TAP;

/*
 * Get the gpio descriptor, set the gpio pin direction to input,
 * and store it in the private structure
 */

```

```

ac->gpio = devm_gpiod_get_index(dev, ADXL345_GPIO_NAME, 0, GPIOD_IN);

/* Get the Linux IRQ number associated with this gpio descriptor */
ac->irq = gpiod_to_irq(ac->gpio);

/* Request threaded interrupt */
devm_request_threaded_irq(input_dev->dev.parent,
                          ac->irq, NULL,
                          adxl345_irq,
                          IRQF_TRIGGER_HIGH | IRQF_ONESHOT,
                          dev_name(dev), ac);

/* Create a group of sysfs entries */
sysfs_create_group(&dev->kobj, &adxl345_attr_group);

/* Register the input device to the input core */
input_register_device(input_dev);

/* Initialize the ADXL345 registers */

/* Set the tap threshold and duration */
AC_WRITE(ac, THRESH_TAP, pdata->tap_threshold);
AC_WRITE(ac, DUR, pdata->tap_duration);

/* Set the axis where the tap will be detected (AXIS Z) */
AC_WRITE(ac, TAP_AXES, pdata->tap_axis_control);

/*
 * Set the data rate and the axis reading power
 * mode. Choose less or higher noise reducing power
 */
AC_WRITE(ac, BW_RATE, RATE(ac->pdata.data_rate) |
        (pdata->low_power_mode ? LOW_POWER : 0));

/* 13-bit full resolution right justified */
AC_WRITE(ac, DATA_FORMAT, pdata->data_range);

/* Set the FIFO mode, no FIFO by default */
AC_WRITE(ac, FIFO_CTL, FIFO_MODE(pdata->fifo_mode) | SAMPLES(pdata->watermark));

/* Map all INTs to INT1 pin */
AC_WRITE(ac, INT_MAP, 0);

/* Enables interrupts */
AC_WRITE(ac, INT_ENABLE, ac->int_mask);

/* Set RUN mode */
AC_WRITE(ac, POWER_CTL, PCTL_MEASURE);

/* Return initialized private structure */
return ac;
}

```

8. You will write a threaded interrupt handler to service the single tap interrupt. In a threaded interrupt, the interrupt handler is executed inside a thread. It is allowed to block during the execution of the interrupt handler, which is often needed to communicate with

an SPI or I2C device inside the handler. In this interrupt handler, you will communicate via SPI with the ADXL345 device. See below the code of the handler:

```
static irqreturn_t adxl345_irq(int irq, void *handle)
{
    struct adxl345 *ac = handle;
    struct adxl345_platform_data *pdata = &ac->pdata;
    int int_stat, tap_stat;

    /*
     * ACT_TAP_STATUS should be read before clearing the interrupt.
     * Avoid reading ACT_TAP_STATUS in case TAP detection is disabled.
     * Read the ACT_TAP_STATUS if any of the axis has been enabled
     */
    if (pdata->tap_axis_control & (TAP_X_EN | TAP_Y_EN | TAP_Z_EN))
        tap_stat = AC_READ(ac, ACT_TAP_STATUS);
    else
        tap_stat = 0;

    /* Read the INT_SOURCE (0x30) register. The interrupt is cleared */
    int_stat = AC_READ(ac, INT_SOURCE);

    /*
     * If the SINGLE_TAP event has occurred, the axl345_do_tap function
     * is called with the ACT_TAP_STATUS register as an argument
     */
    if (int_stat & (SINGLE_TAP)) {
        dev_info(ac->dev, "single tap interrupt has occurred\n");
        adxl345_do_tap(ac, pdata, tap_stat);
    }

    input_sync(ac->input);
    return IRQ_HANDLED;
}
```

9. You will generate the event type EV_KEY with 3 different event codes that will be set depending on the axis where the tap motion detection has been selected. You will send these events in the ISR by calling the `adxl345_do_tap()` function.

```
/*
 * Set EV_KEY type event with 3 events code support.
 * The event is sent when a single tap interrupt is triggered
 */
__set_bit(EV_KEY, input_dev->evbit);
__set_bit(pdata->ev_code_tap[ADXL_X_AXIS], input_dev->keybit);
__set_bit(pdata->ev_code_tap[ADXL_Y_AXIS], input_dev->keybit);
__set_bit(pdata->ev_code_tap[ADXL_Z_AXIS], input_dev->keybit);

static void adxl345_send_key_events(struct adxl345 *ac,
                                   struct adxl345_platform_data *pdata, int status, int press)
{
```

```

    int i;

    for (i = ADXL_X_AXIS; i <= ADXL_Z_AXIS; i++) {
        if (status & (1 << (ADXL_Z_AXIS - i)))
            input_report_key(ac->input, pdata->ev_code_tap[i], press);
    }

    /* Function called in the ISR when there is a SINGLE_TAP event */
    static void adxl345_do_tap(struct adxl345 *ac,
                              struct adxl345_platform_data *pdata,
                              int status)
    {
        adxl345_send_key_events(ac, pdata, status, true);
        input_sync(ac->input);
        adxl345_send_key_events(ac, pdata, status, false);
    }

```

10. You will create several sysfs entries to access the driver from user space. You can set and read the sample rate, read the data of the three axes values, and show the last stored values of the axes by using sysfs hooks.

You will create the sysfs attributes with the `DEVICE_ATTR(name, mode, show, store)` macro:

```

static DEVICE_ATTR(rate, 0664, adxl345_rate_show, adxl345_rate_store);
static DEVICE_ATTR(position, S_IRUGO, adxl345_position_show, NULL);
static DEVICE_ATTR(read, S_IRUGO, adxl345_position_read, NULL);

```

These attributes can be organized into a group as follows:

```

static struct attribute *adxl345_attributes[] = {
    &dev_attr_rate.attr,
    &dev_attr_position.attr,
    &dev_attr_read.attr,
    NULL
};

static const struct attribute_group adxl345_attr_group = {
    .attrs = adxl345_attributes,
};

```

See below the code of the `adxl345_position_read()` function, which will read the data of the three axes:

```

static ssize_t adxl345_position_read(struct device *dev,
                                     struct device_attribute *attr,
                                     char *buf)
{
    struct axis_triple axis;
    ssize_t count;
    struct adxl345 *ac = dev_get_drvdata(dev);
    adxl345_get_triple(ac, &axis);
}

```

```

        count = sprintf(buf, "(%d, %d, %d)\n", axis.x, axis.y, axis.z);

    return count;
}

```

The `adxl345_position_read()` function calls `adxl345_get_triple()`, which in turn calls the `ac->bops->read_block()` function:

```

/* Get the adxl345 axis data */
static void adxl345_get_triple(struct adxl345 *ac, struct axis_triple *axis)
{
    __le16 buf[3];

    ac->bops->read_block(ac->dev, DATA0, DATA1 - DATA0 + 1, buf);
    ac->saved.x = sign_extend32(le16_to_cpu(buf[0]), 12);
    axis->x = ac->saved.x;

    ac->saved.y = sign_extend32(le16_to_cpu(buf[1]), 12);
    axis->y = ac->saved.y;

    ac->saved.z = sign_extend32(le16_to_cpu(buf[2]), 12);
    axis->z = ac->saved.z;
}

```

You can see that `read_block` (a member of the `adxl345_bus_ops` function) is initialized to the `adxl345_spi_read_block` bus function:

```

static const struct adxl345_bus_ops adxl345_spi_bops = {
    .bustype      = BUS_SPI,
    .write        = adxl345_spi_write,
    .read         = adxl345_spi_read,
    .read_block   = adxl345_spi_read_block,
};

```

See below the code of the `adxl345_spi_read_block()` function. The `reg` parameter is the address of the first register you want to read, and `count` is the total number of registers that you will read, starting from the `reg` one. The `buf` parameter is a pointer to the buffer where the values of the axes will be stored.

```

/* Read multiple registers */
static int adxl345_spi_read_block(struct device *dev,
                                unsigned char reg,
                                int count,
                                void *buf)
{
    struct spi_device *spi = to_spi_device(dev);
    ssize_t status;

    /* Add MB flags to the reading */
    reg = ADXL345_READMB_CMD(reg);
}

```

```

/*
 * Write byte stored in reg (address with MB),
 * read count bytes (from successive addresses),
 * and stores them to buf
 */
status = spi_write_then_read(spi, &reg, 1, buf, count);
return (status < 0) ? status : 0;
}

```

The `adxl345_spi_read_block()` function calls `spi_write_then_read()`, which sends to the SPI bus a command byte composed by the address of the first register to read (bits A0 to A5) plus the MB bit (set to one for multi reading) and R bit (set to one for reading), then reads the value of six registers (count), starting from the reg (bits A0 to A5) one.

See below the macros for the SPI commands used to read from and write to your SPI device:

```

#define ADXL345_CMD_MULTB          (1 << 6)
#define ADXL345_CMD_READ           (1 << 7)
#define ADXL345_WRITECMD(reg)      (reg & 0x3F)
#define ADXL345_READCMD(reg)       (ADXL345_CMD_READ | (reg & 0x3F))
#define ADXL345_READMB_CMD(reg)    (ADXL345_CMD_READ | ADXL345_CMD_MULTB \
                                     | (reg & 0x3F))

```

11. Declare a list of devices supported by the driver:

```

static const struct of_device_id adxl345_dt_ids[] = {
    { .compatible = "arrow,adxl345", },
    { }
};
MODULE_DEVICE_TABLE(of, adxl345_dt_ids);

```

12. Define an array of `spi_device_id` structures:

```

static const struct spi_device_id adxl345_id[] = {
    { .name = "adxl345", },
    { }
};
MODULE_DEVICE_TABLE(spi, adxl345_id);

```

13. Add an `spi_driver` structure that will be registered to the SPI bus:

```

static struct spi_driver adxl345_driver = {
    .driver = {
        .name = "adxl345",
        .owner = THIS_MODULE,
        .of_match_table = adxl345_dt_ids,
    },
    .probe = adxl345_spi_probe,
    .remove = adxl345_spi_remove,
    .id_table = adxl345_id,
};

```

14. Register your driver with the SPI bus:

```
module_spi_driver(adxl345_driver);
```

15. Create a new `adxl345_rpi3.c` file in the `linux_5.4_rpi3_drivers` folder, and add `adxl345_rpi3.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

16. Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

17. Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 10-6: `adxl345_rpi3.c`

```
#include <linux/input.h>
#include <linux/module.h>
#include <linux/spi/spi.h>
#include <linux/of_gpio.h>
#include <linux/spi/spi.h>
#include <linux/interrupt.h>

#define ADXL345_CMD_MULTB      (1 << 6)
#define ADXL345_CMD_READ      (1 << 7)
#define ADXL345_WRITECMD(reg) (reg & 0x3F)
#define ADXL345_READCMD(reg)  (ADXL345_CMD_READ | (reg & 0x3F))
#define ADXL345_READMB_CMD(reg) (ADXL345_CMD_READ | ADXL345_CMD_MULTB \
| (reg & 0x3F))

/* ADXL345 Register Map */
#define DEVID                0x00 /* R Device ID */
#define THRESH_TAP           0x1D /* R/W Tap threshold */
#define DUR                  0x21 /* R/W Tap duration */
#define TAP_AXES             0x2A /* R/W Axis control for tap/double tap */
#define ACT_TAP_STATUS       0x2B /* R Source of tap/double tap */
#define BW_RATE              0x2C /* R/W Data rate and power mode control */
#define POWER_CTL            0x2D /* R/W Power saving features control */
#define INT_ENABLE           0x2E /* R/W Interrupt enable control */
#define INT_MAP              0x2F /* R/W Interrupt mapping control */
#define INT_SOURCE           0x30 /* R Source of interrupts */
#define DATA_FORMAT         0x31 /* R/W Data format control */
#define DATA0               0x32 /* R X-Axis Data 0 */
#define DATA1               0x33 /* R X-Axis Data 1 */
#define DATA0               0x34 /* R Y-Axis Data 0 */
#define DATA1               0x35 /* R Y-Axis Data 1 */
#define DATA0               0x36 /* R Z-Axis Data 0 */
#define DATA1               0x37 /* R Z-Axis Data 1 */
```



```

#define FIFO_CTL                0x38    /* R/W FIFO control */

/* DEVIDs */
#define ID_ADXL345              0xE5

/* INT_ENABLE/INT_MAP/INT_SOURCE Bits */
#define SINGLE_TAP              (1 << 6)

/* TAP_AXES Bits */
#define TAP_X_EN                (1 << 2)
#define TAP_Y_EN                (1 << 1)
#define TAP_Z_EN                (1 << 0)

/* BW_RATE Bits */
#define LOW_POWER               (1 << 4)
#define RATE(x)                 ((x) & 0xF)

/* POWER_CTL Bits */
#define PCTL_MEASURE             (1 << 3)
#define PCTL_STANDBY            0X00

/* DATA_FORMAT Bits */
#define FULL_RES                (1 << 3)

/* FIFO_CTL Bits */
#define FIFO_MODE(x)            (((x) & 0x3) << 6)
#define FIFO_BYPASS             0
#define FIFO_FIFO               1
#define FIFO_STREAM             2
#define SAMPLES(x)              ((x) & 0x1F)

/* FIFO_STATUS Bits */
#define ADXL_X_AXIS             0
#define ADXL_Y_AXIS             1
#define ADXL_Z_AXIS             2

#define ADXL345_GPIO_NAME       "int"

/* Macros to do SPI operations */
#define AC_READ(ac, reg)         ((ac)->bops->read((ac)->dev, reg))
#define AC_WRITE(ac, reg, val)   ((ac)->bops->write((ac)->dev, reg, val))

struct adxl345_bus_ops {
    u16 bustype;
    int (*read)(struct device *, unsigned char);
    int (*read_block)(struct device *, unsigned char, int, void *);
    int (*write)(struct device *, unsigned char, unsigned char);
};

struct axis_triple {
    int x;
    int y;
    int z;
};

```

```

struct adxl345_platform_data {
    /*
     * low_power_mode:
     * A '0' = Normal operation and a '1' = Reduced
     * power operation with somewhat higher noise.
     */

    u8 low_power_mode;

    /*
     * tap_threshold:
     * holds the threshold value for tap detection/interrupts.
     * The data format is unsigned. The scale factor is 62.5 mg/LSB
     * (i.e. 0xFF = +16 g). A zero value may result in undesirable
     * behavior if Tap/Double Tap is enabled.
     */

    u8 tap_threshold;

    /*
     * tap_duration:
     * is an unsigned time value representing the maximum
     * time that an event must be above the tap_threshold threshold
     * to qualify as a tap event. The scale factor is 625 us/LSB. A zero
     * value will prevent Tap/Double Tap functions from working.
     */

    u8 tap_duration;

    /*
     * TAP_X/Y/Z Enable: Setting TAP_X, Y, or Z Enable enables X,
     * Y, or Z participation in Tap detection. A '0' excludes the
     * selected axis from participation in Tap detection.
     * Setting the SUPPRESS bit suppresses Double Tap detection if
     * acceleration greater than tap_threshold is present during the
     * tap_latency period, i.e. after the first tap but before the
     * opening of the second tap window.
     */

#define ADXL_TAP_X_EN(1 << 2)
#define ADXL_TAP_Y_EN(1 << 1)
#define ADXL_TAP_Z_EN(1 << 0)

    u8 tap_axis_control;

    /*
     * data_rate:
     * Selects device bandwidth and output data rate.
     * RATE = 3200 Hz / (2^(15 - x)). Default value is 0x0A, or 100 Hz
     * Output Data Rate. An Output Data Rate should be selected that
     * is appropriate for the communication protocol and frequency
     * selected. Selecting too high of an Output Data Rate with a low
     * communication speed will result in samples being discarded.

```

```

    */

    u8 data_rate;

    /*
    * data_range:
    * FULL_RES: When this bit is set with the device is
    * in Full-Resolution Mode, where the output resolution increases
    * with RANGE to maintain a 4 mg/LSB scale factor. When this
    * bit is cleared the device is in 10-bit Mode and RANGE determine the
    * maximum g-Range and scale factor.
    */

#define ADXL_FULL_RES                (1 << 3)
#define ADXL_RANGE_PM_2g             0
#define ADXL_RANGE_PM_4g             1
#define ADXL_RANGE_PM_8g             2
#define ADXL_RANGE_PM_16g            3

    u8 data_range;

    /*
    * A valid BTN or KEY Code; use tap_axis_control to disable
    * event reporting
    */

    u32 ev_code_tap[3];

    /*
    * fifo_mode:
    * BYPASS The FIFO is bypassed
    * FIFO FIFO collects up to 32 values then stops collecting data
    * STREAM FIFO holds the last 32 data values. Once full, the FIFO's
    * oldest data is lost as it is replaced with newer data
    * DEFAULT should be FIFO_STREAM
    */

    u8 fifo_mode;

    /*
    * watermark:
    * The Watermark feature can be used to reduce the interrupt load
    * of the system. The FIFO fills up to the value stored in watermark
    * [1..32] and then generates an interrupt.
    * A '0' disables the watermark feature.
    */

    u8 watermark;

};

/* Set initial adxl345 register values */
static const struct adxl345_platform_data adxl345_default_init = {
    .tap_threshold = 50,

```

```

        .tap_duration = 3,
        .tap_axis_control = ADXL_TAP_Z_EN,
        .data_rate = 8,
        .data_range = ADXL_FULL_RES,
        .ev_code_tap = {BTN_TOUCH, BTN_TOUCH, BTN_TOUCH}, /* EV_KEY {x,y,z} */
        .fifo_mode = FIFO_BYPASS,
        .watermark = 0,
};

/* Create private data structure */
struct adxl345 {
    struct gpio_desc *gpio;
    struct device *dev;
    struct input_dev *input;
    struct adxl345_platform_data pdata;
    struct axis_triple saved;
    u8 phys[32];
    int irq;
    u32 model;
    u32 int_mask;
    const struct adxl345_bus_ops *bops;
};

/* Get the adxl345 axis data */
static void adxl345_get_triple(struct adxl345 *ac, struct axis_triple *axis)
{
    __le16 buf[3];

    ac->bops->read_block(ac->dev, DATA0, DATA1 - DATA0 + 1, buf);

    ac->saved.x = sign_extend32(le16_to_cpu(buf[0]), 12);
    axis->x = ac->saved.x;

    ac->saved.y = sign_extend32(le16_to_cpu(buf[1]), 12);
    axis->y = ac->saved.y;

    ac->saved.z = sign_extend32(le16_to_cpu(buf[2]), 12);
    axis->z = ac->saved.z;
}

/*
 * This function is called inside adxl34x_do_tap() in the ISR
 * when there is a SINGLE_TAP event. The function checks
 * the TAP_X, TAP_Y and TAP_Z bits of the ACT_TAP_STATUS (0x2B), starting
 * from the TAP_X source bit. If the axis is involved in the event
 * there is a EV_KEY event
 */
static void adxl345_send_key_events(struct adxl345 *ac,
                                   struct adxl345_platform_data *pdata,
                                   int status, int press)
{
    int i;

    for (i = ADXL_X_AXIS; i <= ADXL_Z_AXIS; i++) {

```

```

        if (status & (1 << (ADXL_Z_AXIS - i)))
            input_report_key(ac->input, pdata->ev_code_tap[i], press);
    }
}

/* Function called in the ISR when there is a SINGLE_TAP event */
static void adxl345_do_tap(struct adxl345 *ac,
                          struct adxl345_platform_data *pdata,
                          int status)
{
    adxl345_send_key_events(ac, pdata, status, true);
    input_sync(ac->input);
    adxl345_send_key_events(ac, pdata, status, false);
}

/* Interrupt service routine */
static irqreturn_t adxl345_irq(int irq, void *handle)
{
    struct adxl345 *ac = handle;
    struct adxl345_platform_data *pdata = &ac->pdata;
    int int_stat, tap_stat;

    /*
     * ACT_TAP_STATUS should be read before clearing the interrupt
     * Avoid reading ACT_TAP_STATUS in case TAP detection is disabled
     * Read the ACT_TAP_STATUS if any of the axis has been enabled
     */
    if (pdata->tap_axis_control & (TAP_X_EN | TAP_Y_EN | TAP_Z_EN))
        tap_stat = AC_READ(ac, ACT_TAP_STATUS);
    else
        tap_stat = 0;

    /* Read the INT_SOURCE (0x30) register. The interrupt is cleared */
    int_stat = AC_READ(ac, INT_SOURCE);

    /*
     * If the SINGLE_TAP event has occurred, the axl345_do_tap function
     * is called with the ACT_TAP_STATUS register as an argument
     */
    if (int_stat & (SINGLE_TAP)) {
        dev_info(ac->dev, "single tap interrupt has occurred\n");
        adxl345_do_tap(ac, pdata, tap_stat);
    };

    input_sync(ac->input);

    return IRQ_HANDLED;
}

static ssize_t adxl345_rate_show(struct device *dev,
                                struct device_attribute *attr,
                                char *buf)
{
    struct adxl345 *ac = dev_get_drvdata(dev);

```

```

    return sprintf(buf, "%u\n", RATE(ac->pdata.data_rate));
}

static ssize_t adxl345_rate_store(struct device *dev, struct device_attribute *attr,
                                const char *buf, size_t count)
{
    struct adxl345 *ac = dev_get_drvdata(dev);
    u8 val;
    int error;

    /* Transform char array to u8 value */
    error = kstrtou8(buf, 10, &val);
    if (error)
        return error;

    /*
     * If you set ac->pdata.low_power_mode = 1,
     * then lower power mode but higher noise is selected,
     * getting LOW_POWER macro. By default ac->pdata.low_power_mode = 0
     * RATE(val) sets to 0 the 4 upper u8 bits
     */
    ac->pdata.data_rate = RATE(val);
    AC_WRITE(ac, BW_RATE, ac->pdata.data_rate | (ac->pdata.low_power_mode ? LOW_POWER : 0));

    return count;
}

static DEVICE_ATTR(rate, 0664, adxl345_rate_show, adxl345_rate_store);

static ssize_t adxl345_position_show(struct device *dev,
                                    struct device_attribute *attr,
                                    char *buf)
{
    struct adxl345 *ac = dev_get_drvdata(dev);
    ssize_t count;
    count = sprintf(buf, "(%d, %d, %d)\n", ac->saved.x, ac->saved.y, ac->saved.z);

    return count;
}

static DEVICE_ATTR(position, S_IRUGO, adxl345_position_show, NULL);

static ssize_t adxl345_position_read(struct device *dev,
                                    struct device_attribute *attr,
                                    char *buf)
{
    struct axis_triple axis;
    ssize_t count;
    struct adxl345 *ac = dev_get_drvdata(dev);
    adxl345_get_triple(ac, &axis);
    count = sprintf(buf, "(%d, %d, %d)\n", axis.x, axis.y, axis.z);

    return count;
}

static DEVICE_ATTR(read, S_IRUGO, adxl345_position_read, NULL);

```

```

static struct attribute *adx1345_attributes[] = {
    &dev_attr_rate.attr,
    &dev_attr_position.attr,
    &dev_attr_read.attr,
    NULL
};

static const struct attribute_group adx1345_attr_group = {
    .attrs = adx1345_attributes,
};

struct adx1345 *adx1345_probe(struct device *dev, const struct adx1345_bus_ops *bops)
{
    /* Declare your private structure */
    struct adx1345 *ac;

    /* Create the input device */
    struct input_dev *input_dev;

    /* Create pointer to const struct platform data */
    const struct adx1345_platform_data *pdata;
    int err;
    u8 revid;

    /* Allocate private structure*/
    ac = devm_kzalloc(dev, sizeof(*ac), GFP_KERNEL);
    if (!ac) {
        dev_err(dev, "Failed to allocate memory\n");
        err = -ENOMEM;
        goto err_out;
    }

    /* Allocate the input_dev structure */
    input_dev = devm_input_allocate_device(dev);
    if (!ac || !input_dev) {
        dev_err(dev, "failed to allocate input device\n");
        err = -ENOMEM;
        goto err_out;
    }

    /* Initialize your private structure */

    /*
     * Store the previously initialized platform data
     * in your private structure
     */
    pdata = &adx1345_default_init; /* Points to const platform data */
    ac->pdata = *pdata; /* Store values to pdata inside ac */
    pdata = &ac->pdata; /* Change where pdata points, now to pdata in private ac */

    ac->input = input_dev;
    ac->dev = dev; /* dev is &spi->dev */

```

```

/* Store the SPI operations in your private structure */
ac->bops = bops;

revid = AC_READ(ac, DEVID);
dev_info(dev, "DEVID: %d\n", revid);

if (revid == 0xE5) {
    dev_info(dev, "ADXL345 is found");
}
else
{
    dev_err(dev, "Failed to probe %s\n", input_dev->name);
    err = -ENODEV;
    goto err_out;
}

snprintf(ac->phys, sizeof(ac->phys), "%s/input0", dev_name(dev));

/* Initialize the input device */
input_dev->name = "ADXL345 accelerometer";
input_dev->phys = ac->phys;
input_dev->dev.parent = dev;
input_dev->id.product = ac->model;
input_dev->id.bustype = bops->bustype;

/* Attach the input device and the private structure */
input_set_drvdata(input_dev, ac);

/*
 * Set EV_KEY type event with 3 events code support.
 * The event is sent when a single tap interrupt is triggered
 */
__set_bit(EV_KEY, input_dev->evbit);
__set_bit(pdata->ev_code_tap[ADXL_X_AXIS], input_dev->keybit);
__set_bit(pdata->ev_code_tap[ADXL_Y_AXIS], input_dev->keybit);
__set_bit(pdata->ev_code_tap[ADXL_Z_AXIS], input_dev->keybit);

/*
 * Check if any of the axis has been enabled
 * and set the interrupt mask.
 * In this driver only is enabled the SINGLE_TAP interrupt
 */
if (pdata->tap_axis_control & (TAP_X_EN | TAP_Y_EN | TAP_Z_EN))
    ac->int_mask |= SINGLE_TAP;

/*
 * Get the gpio descriptor, set the gpio pin direction to input,
 * and store it in the private structure
 */
ac->gpio = devm_gpiod_get_index(dev, ADXL345_GPIO_NAME, 0, GPIOD_IN);
if (IS_ERR(ac->gpio)) {
    dev_err(dev, "gpio get index failed\n");
    err = PTR_ERR(ac->gpio);
    goto err_out;
}

```



```

}

/* Get the Linux IRQ number associated with this gpio descriptor */
ac->irq = gpiod_to_irq(ac->gpio);
if (ac->irq < 0) {
    dev_err(dev, "gpio get irq failed\n");
    err = ac->irq;
    goto err_out;
}
dev_info(dev, "The IRQ number is: %d\n", ac->irq);

/* Request threaded interrupt */
err = devm_request_threaded_irq(input_dev->dev.parent, ac->irq, NULL,
                                adxl345_irq, IRQF_TRIGGER_HIGH | IRQF_ONESHOT,
                                dev_name(dev), ac);

if (err)
    goto err_out;

/* Create a group of sysfs entries */
err = sysfs_create_group(&dev->kobj, &adxl345_attr_group);
if (err)
    goto err_out;

/* Register the input device to the input core */
err = input_register_device(input_dev);
if (err)
    goto err_remove_attr;

/* Initialize the ADXL345 registers */

/* Set the tap threshold and duration */
AC_WRITE(ac, THRESH_TAP, pdata->tap_threshold);
AC_WRITE(ac, DUR, pdata->tap_duration);

/* Set the axis where the tap will be detected */
AC_WRITE(ac, TAP_AXES, pdata->tap_axis_control);

/*
 * Set the data rate
 * and power mode (higher noise, less power)
 */
AC_WRITE(ac, BW_RATE, RATE(ac->pdata.data_rate) | (pdata->low_power_mode ? LOW_POWER : 0));

/* 13-bit full resolution right justified */
AC_WRITE(ac, DATA_FORMAT, pdata->data_range);

/* Set the FIFO mode, no FIFO by default */
AC_WRITE(ac, FIFO_CTL, FIFO_MODE(pdata->fifo_mode) | SAMPLES(pdata->watermark));

/* Map all INTs to INT1 pin */
AC_WRITE(ac, INT_MAP, 0);

/* Enables interrupts */
AC_WRITE(ac, INT_ENABLE, ac->int_mask);

```

```

    /* Set RUN mode */
    AC_WRITE(ac, POWER_CTL, PCTL_MEASURE);

    /* Return initialized private structure */
    return ac;

err_remove_attr:
    sysfs_remove_group(&dev->kobj, &adx1345_attr_group);

/*
 * This function returns a pointer
 * to a struct ac or an err pointer
 */
err_out:
    return ERR_PTR(err);
}

/* Read the value of the register */
static int adxl345_spi_read(struct device *dev, unsigned char reg)
{
    struct spi_device *spi = to_spi_device(dev);
    u8 cmd;
    cmd = ADXL345_READCMD(reg);
    return spi_w8r8(spi, cmd);
}

/*
 * Write 2 bytes, the address
 * of the register and the value to store on it
 */
static int adxl345_spi_write(struct device *dev, unsigned char reg, unsigned char val)
{
    struct spi_device *spi = to_spi_device(dev);
    u8 buf[2];

    buf[0] = ADXL345_WRITECMD(reg);
    buf[1] = val;

    return spi_write(spi, buf, sizeof(buf));
}

/* Read multiple registers */
static int adxl345_spi_read_block(struct device *dev,
                                   unsigned char reg,
                                   int count, void *buf)
{
    struct spi_device *spi = to_spi_device(dev);
    ssize_t status;

    /* Add MB flags to the reading */
    reg = ADXL345_READMB_CMD(reg);

    /*

```

```

    * Write byte stored in reg (address with MB),
    * read count bytes (from successive addresses),
    * and stores them to buf
    */
    status = spi_write_then_read(spi, &reg, 1, buf, count);

    return (status < 0) ? status : 0;
}

/* Initialize struct adxl345_bus_ops to SPI bus functions */
static const struct adxl345_bus_ops adxl345_spi_bops = {
    .bustype= BUS_SPI,
    .write    = adxl345_spi_write,
    .read     = adxl345_spi_read,
    .read_block = adxl345_spi_read_block,
};

static int adxl345_spi_probe(struct spi_device *spi)
{
    struct adxl345 *ac;

    /* Send the spi operations */
    ac = adxl345_probe(&spi->dev, &adxl345_spi_bops);

    if (IS_ERR(ac))
        return PTR_ERR(ac);

    /* Attach the SPI device to the private structure */
    spi_set_drvdata(spi, ac);

    return 0;
}

static int adxl345_spi_remove(struct spi_device *spi)
{
    struct adxl345 *ac = spi_get_drvdata(spi);
    dev_info(ac->dev, "my_remove() function is called.\n");
    sysfs_remove_group(&ac->dev->kobj, &adxl345_attr_group);
    input_unregister_device(ac->input);
    AC_WRITE(ac, POWER_CTL, PCTL_STANDBY);
    dev_info(ac->dev, "unregistered accelerometer\n");
    return 0;
}

static const struct of_device_id adxl345_dt_ids[] = {
    { .compatible = "arrow,adxl345", },
    { }
};

MODULE_DEVICE_TABLE(of, adxl345_dt_ids);

```

```
static const struct spi_device_id adxl345_id[] = {
    { .name = "adxl345", },
    { }
};
MODULE_DEVICE_TABLE(spi, adxl345_id);

static struct spi_driver adxl345_driver = {
    .driver = {
        .name = "adxl345",
        .owner = THIS_MODULE,
        .of_match_table = adxl345_dt_ids,
    },
    .probe = adxl345_spi_probe,
    .remove = adxl345_spi_remove,
    .id_table = adxl345_id,
};
module_spi_driver(adxl345_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("ADXL345 Three-Axis Accelerometer SPI Bus Driver");
```

adxl345_rpi3.ko demonstration

Disable the Device Tree overlay for the CY8C9520A device if you have executed the LAB 7.6:

```
root@raspberrypi:/home/pi# cd /boot/
root@raspberrypi:/boot# nano config.txt
# Uncomment some or all of these to enable the optional hardware interfaces
dtparam=i2c_arm=on
#dtparam=i2s=on
dtparam=spi=on
dtoverlay=spi0-cs
enable_uart=1
kernel=kernel7.img
#dtoverlay=cy8c9520a
#dtoverlay=i2c1,pins_2_3
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Load the adxl345_rpi3.ko module:

```
root@raspberrypi:/home/pi# insmod adxl345_rpi3.ko
adxl345_rpi3: loading out-of-tree module taints kernel.
adxl345 spi0.0: DEVID: 229
adxl345 spi0.0: ADXL345 is found
adxl345 spi0.0: The IRQ number is: 166
input: ADXL345 accelerometer as /devices/platform/soc/3f204000.sp
i/spi_master/spi0/spi0.0/input/input0
```

See the entries under the ADXL345 input device:

```
root@raspberrypi:/sys/class/input/input0/device# ls
driver      input      of_node    power      read       subsystem
driver_override  modalias  position   rate       statistics uevent
```

Change the position of the ADXL345 board, and read the values of the three axes:

```
root@raspberrypi:/sys/class/input/input0/device# cat read
(185, -188, 61)
```

```
root@raspberrypi:/sys/class/input/input0/device# cat read
(-126, 70, 191)
```

Read the current data rate:

```
root@raspberrypi:/sys/class/input/input0/device# cat rate
8
```

Change the data rate:

```
root@raspberrypi:/sys/class/input/input0/device# echo 10 > rate
root@raspberrypi:/sys/class/input/input0/device# cat rate
10
```

Launch the evtest application. Move the accelerometer board in the z axis direction, and see the interrupts and events generated:

```
root@raspberrypi:/sys/class/input/input0/device# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      ADXL345 accelerometer
Select the device event number [0-0]: 0
Input driver version is 1.0.1
Input device ID: bus 0x1c vendor 0x0 product 0x0 version 0x0
Input device name: "ADXL345 accelerometer"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 330 (BTN_TOUCH)
Properties:
Testing ... (interrupt to exit)
adxl345 spi0.0: single tap interrupt has occurred
Event: time 1617907140.493226, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 1
Event: time 1617907140.493226, ----- SYN_REPORT -----
Event: time 1617907140.493239, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 0
Event: time 1617907140.493239, ----- SYN_REPORT -----
Event: time 1617907140.499904, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 1
Event: time 1617907140.499904, ----- SYN_REPORT -----
Event: time 1617907140.499912, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 0
Event: time 1617907140.499912, ----- SYN_REPORT -----
adxl345 spi0.0: single tap interrupt has occurred
Event: time 1617907142.362432, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 1
Event: time 1617907142.362432, ----- SYN_REPORT -----
Event: time 1617907142.362446, type 1 (EV_KEY), code 330 (BTN_TOUCH), value 0
Event: time 1617907142.362446, ----- SYN_REPORT -----
```

Exit with ^C:

```
root@raspberrypi:/sys/class/input/input0/device# cd /home/pi
```

Remove the `adxl345_rpi3.ko` module:

```
root@raspberrypi:/home/pi# rmmod adxl345_rpi3.ko
adxl345 spi0.0: my_remove() function is called.
adxl345 spi0.0: unregistered accelerometer
```

11

Industrial I/O Subsystem

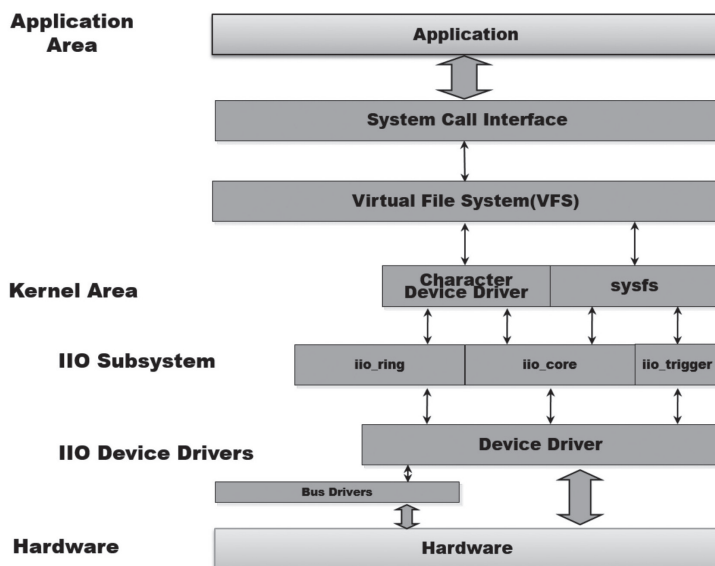
IIO (Industrial I/O) is a subsystem that was created to support Analog to Digital Converters (ADCs), Digital to Analog Converters (DACs) and various types of sensors. The IIO subsystem can be used from user space (through the `libiio` library and the IIO Linux kernel tools) and from kernel space using the IIO kernel API.

These are some examples of sensors supported in IIO:

- Analog to digital converters (ADCs)
- Accelerometers
- Capacitance to digital converters (CDCs)
- Digital to analog converters (DACs)
- Gyroscopes
- Inertial measurement units (IMUs)
- Color and light sensors
- Magnetometers
- Pressure sensors
- Proximity sensors
- Temperature sensors

Usually, these sensors are connected via SPI or I2C. A common use case of the sensor devices is to have combined functionality (e.g., light plus proximity sensor). However, typical DMA mastered devices such as ones connected to a high speed synchronous serial or high speed synchronous parallel peripherals are also subject to this subsystem.

Note: The Industrial I/O subsystem is explained in detail in the Linux driver implementer's API guide (<https://www.kernel.org/doc/html/latest/driver-api/iio/index.html>). Some parts of that documentation have been extracted to be used in the following sections.



The Industrial I/O core offers:

1. A unified framework for writing drivers for many different types of embedded sensors.
2. A standard interface to user space applications manipulating sensors.

The implementation can be found under `linux/drivers/iio/` folder in the `industrialio-*` files. An IIO device usually corresponds to a single hardware sensor, and it provides all the information needed by a driver handling a device. First, you will have a look at the functionality embedded in an IIO device, then you will see how a device driver makes use of an IIO device.

The IIO framework provides several interfaces:

1. `/sys/bus/iio/iio:deviceX` -- This represents a hardware sensor and groups together the data channels of the same chip. It is used to read and write data directly at low rates.
2. `/dev/iio:deviceX` -- This is the character device node interface used to output events and sensor data. The standard file API (`open()`, `read()`, `write()`, etc.) allows access to it.

A typical IIO driver will register itself as an I2C or SPI driver, and it will create two routines: `probe()` and `remove()`. At `probe()`:

1. The driver will call `devm_iio_device_alloc()`, which allocates memory for an IIO device.
2. The driver will initialize the fields of the IIO device with driver specific information (e.g., device name, device channels).

3. The driver will call `devm_iio_device_register()`, which registers the device to the IIO core. Now, the device is global to the rest of the driver functions until it is unregistered. After this call, the device is ready to accept requests from user space applications.

IIO device sysfs interface

Attributes are sysfs files used to configure events and data that should come out of the device and to expose device info. For a device with index `X`, the attributes can be found under the `/sys/bus/iio/iio:deviceX/` directory. These are some common attributes:

- `name` -- Description of the physical chip.
- `dev` -- Shows the major:minor pair associated with `/dev/iio:deviceX` node.
- Device configuration attributes like `sampling_frequency_available`.
- Data channel access attributes like `out_voltage0_raw`.
- Attributes under `buffer/`, `events/`, `trigger/` and `scan elements/` subdirectories.

IIO device channels

An IIO device channel is a representation of a data channel. An IIO device can have one or multiple channels. For example:

- A thermometer sensor has one channel representing the temperature measurement.
- A light sensor with two channels indicating the measurements in the visible and infrared spectrum.
- An accelerometer can have up to three channels representing acceleration on the X, Y and Z axes.

An IIO channel is described by the `iio_chan_spec` structure.

Channel sysfs attributes exposed to user space are specified in the form of bitmasks. Depending on their specific or shared info, attributes can be set in one of the following masks:

- **`info_mask_separate`** attributes will be specific to this channel.
- **`info_mask_shared_by_type`** attributes are shared by all channels of the same type.
- **`info_mask_shared_by_dir`** attributes are shared by all channels of the same direction.
- **`info_mask_shared_by_all`** attributes are shared by all channels.

When there are multiple data channels per channel type, there are two ways to distinguish between them:

1. Set the `.modified` field of `iio_chan_spec` to 1. Modifiers are specified by using the `.channel2` field of the same `iio_chan_spec` structure and are used to indicate a physically unique characteristic of the channel, such as its direction or spectral response. For example, a light

sensor can have two channels, one for infrared light and one for both infrared and visible light.

2. Set the `.indexed` field of `iio_chan_spec` to 1. In this case, the channel is simply another instance with an index specified by the `.channel` field.

The IIO channel definitions will generate data channel access attributes, as the ones of the example below:

```
/sys/bus/iio/devices/iio:deviceX/out_voltage0_raw
/sys/bus/iio/devices/iio:deviceX/out_voltage1_raw
/sys/bus/iio/devices/iio:deviceX/out_voltage2_raw
```

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 `const iio_direction` char pointer array located 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 `const iio_chan_type_name_spec` char pointer array:

```
static const char * const iio_chan_type_name_spec[] = {
    [IIO_VOLTAGE] = "voltage",
    [IIO_CURRENT] = "current",
    [IIO_POWER] = "power",
    [IIO_ACCEL] = "accel",
    [...]
    [IIO_UVINDEK] = "uvindex",
    [IIO_ELECTRICALCONDUCTIVITY] = "electricalconductivity",
    [IIO_COUNT] = "count",
    [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 `const iio_modifier_names` char pointer array:

```
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",
[IIO_MOD_Y_AND_Z] = "y&z",

[...]

[IIO_MOD_CO2] = "co2",
[IIO_MOD_VOC] = "voc",

};

```

- **info_mask** depends on the channel info mask, private or shared, indexing value in the `const iio_chan_info_postfix` char pointer array:

```

/* 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] = "input",
    [IIO_CHAN_INFO_SCALE] = "scale",
    [IIO_CHAN_INFO_CALIBBIAS] = "calibbias",

    [...]

    [IIO_CHAN_INFO_SAMP_FREQ] = "sampling_frequency",    [IIO_CHAN_INFO_FREQUENCY] =
    "frequency",

    [...]

};

```

The iio_info structure

This structure is used to declare the hooks that the core can use for a device. These kernel callbacks will be called for each read and write operation to the sysfs attributes. See below the initialization of a struct `iio_info` for the `adxl345` accelerometer device:

```

static const struct iio_info adxl345_info = {
    .driver_module      = THIS_MODULE,
    .read_raw           = adxl345_read_raw,
    .write_raw          = adxl345_write_raw,
    .read_event_value   = adxl345_read_event,
    .write_event_value  = adxl345_write_event,
};

```

`read_raw` is called to request a value from the IIO device. A bitmask allows to know more precisely which type of value is requested and for which channel if needed. The return value will specify the type of value (`val` or `val2`) returned by the device. The `val` and `val2` values will contain the elements making up the returned value.

`write_raw` is called to write a value to the IIO device. Parameters are the same as for `read_raw`. When you write, for example, a `x` value to the `out_voltage0_raw` sysfs attribute, the `write_raw` hook is called with the `mask` argument set to `IIO_CHAN_INFO_RAW`, the `chan` argument set with the `iio_chan_`

spec structure corresponding to the channel 0 (chan->channel is 0), and the val argument set to the x value.

Buffers

The Industrial I/O core offers a way for continuous data capture based on a trigger source. Multiple data channels can be read at once from /dev/iio:deviceX character device node, thus reducing the CPU load.

IIO buffer sysfs interface

An IIO buffer has associated sysfs attributes under the /sys/bus/iio/iio:deviceX/buffer/ directory. Here are some of the existing attributes:

- length -- Total number of data samples (capacity) that can be stored by the buffer.
- enable -- Activate the buffer capture.

The meta information associated with a channel reading placed in a buffer is called a scan element. The /sys/bus/iio/iio:deviceX/scan_elements/ directory contains type attributes which describe the scan element data storage within the buffer and hence the form in which it is read from user space. Format is [be|le]:[s|u]bits/storagebitsXrepeat[>>shift]:

- be or le specifies big or little endian.
- s or u specifies if signed (2's complement) or unsigned.
- bits is the number of valid data bits.
- storagebits is the number of bits (after padding) that it occupies in the buffer.
- shift is the shift that needs to be applied prior to masking out unused bits.
- repeat specifies the number of bits/storagebits repetitions. When the repeat element is 0 or 1, then the repeat value is omitted.

For example, a driver for a 3-axis accelerometer device of 12 bit resolution, with data stored in two 8-bit registers, will have the following scan element type for each axis:

```
$ cat /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_y_type
le:s12/16>>4
```

A user space application will interpret the data samples, read from the buffer, as two byte little endian signed data that needs a 4 bits right shift before masking out the 12 valid bits of data.

IIO buffer setup

A driver should initialize the following fields (marked in bold below) within an iio_chan_spec structure to implement IIO buffer support:

```

struct iio_chan_spec {
/* other members */
    int scan_index
    struct {
        char sign;
        u8 realbits;
        u8 storagebits;
        u8 shift;
        u8 repeat;
        enum iio_endian endianness;
    } scan_type;
/* other members */
};

```

In the next chapter, you will write an accelerometer driver with the following channel definition that includes IIO buffer support:

```

static const struct iio_chan_spec adxl345_channels[] = {
    ADXL345_CHANNEL(DATA_X0, X, 0),
    ADXL345_CHANNEL(DATA_Y0, Y, 1),
    ADXL345_CHANNEL(DATA_Z0, Z, 2),
    IIO_CHAN_SOFT_TIMESTAMP(3),
};

#define ADXL345_CHANNEL(reg, axis, idx) {
    .type = IIO_ACCEL,
    .modified = 1,
    .channel2 = IIO_MOD_##axis,
    .address = reg,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE) |
                                BIT(IIO_CHAN_INFO_SAMP_FREQ),
    .scan_index = idx,
    .scan_type = {
        .sign = 's',
        .realbits = 13,
        .storagebits = 16,
        .endianness = IIO_LE,
    },
    .event_spec = &adxl345_event,
    .num_event_specs = 1
}

```

Here, `scan_index` defines the order in which the enabled channels are placed inside the buffer. Channels with a lower `scan_index` will be placed before channels with a higher index. Each channel needs to have a unique `scan_index`.

Setting `scan_index` to -1 can be used to indicate that the specific channel does not support buffered capture. In this case, no entries will be created for the channel in the `scan_elements` directory.

The function that will allocate the **trigger buffer** for your device (usually called in the `probe()` function) is `iio_triggered_buffer_setup()`. In the next section, you will see what an IIO trigger is.

The data (i.e., the accelerometer axis values) will be pushed to the IIO device's buffer by using the `iio_push_to_buffers_with_timestamp()` function within the trigger handler. If timestamps are enabled for the device, the function will store the supplied timestamp as the last element in the sample data buffer before pushing it to the device buffers. The sample data buffer needs to be large enough to hold the additional timestamp (usually the buffer should be `indio->scan_bytes` bytes large).

Triggers

In many situations, it is useful for a driver to be able to capture data based on some external event (trigger) instead of periodically polling for data. An IIO trigger can be provided by a device driver that also has an IIO device based on hardware generated events (e.g., data ready or threshold exceeded) or provided by a separate driver from an independent interrupt source (e.g., GPIO line connected to some external system, timer interrupt or user space writing a specific file in `sysfs`). A trigger may initiate data capture for a number of sensors and may be completely unrelated to the sensor itself.

You can develop your own trigger driver, but in this chapter, you will focus only on existing ones. These are:

- **iio-trig-interrupt:** This provides support for using any IRQ as an IIO trigger. The kernel option to enable this trigger mode is `CONFIG_IIO_INTERRUPT_TRIGGER`.
- **iio-trig-hrtimer:** This provides a frequency-based IIO trigger using HRT as the interrupt source. The kernel option responsible for this trigger mode is `IIO_HRTIMER_TRIGGER`.
- **iio-trig-sysfs:** This allows us to use a `sysfs` entry to trigger a data capture. The kernel option responsible for this trigger mode is `CONFIG_IIO_SYSFS_TRIGGER`.

Triggered buffers

Now that you know what buffers and triggers are, let's see how they work together. As it was indicated in the previous section, a trigger buffer is allocated by using the `iio_triggered_buffer_setup()` function. This function combines some common tasks which will normally be performed when setting up a triggered buffer. It will allocate the buffer and the pollfunc. Before calling this function, the `indio_dev` structure should already be completely initialized, but not yet registered. In practice, this means that this function should be called right before `iio_device_register()`. To free the resources allocated by this function, you will call `iio_triggered_buffer_cleanup()`. You can also use the managed functions `devm_iio_triggered_buffer_setup()` and `devm_iio_device_register()`. See a description of the `iio_triggered_buffer_setup()` parameters below:

```
int iio_triggered_buffer_setup(struct iio_dev *indio_dev,
                             irqreturn_t (*h)(int irq, void *p),
                             irqreturn_t (*thread)(int irq, void *p),
```

```
const struct iio_buffer_setup_ops *setup_ops)
```

- `struct iio_dev *indio_dev`: A pointer to the IIO device structure.
- `irqreturn_t (*h)(int irq, void *p)`: A function which will be used as pollfunc top half. It should do as little processing as possible because it runs in interrupt context. The most common operation is the recording of the current timestamp using the `iio_pollfunc_store_time()` function.
- `irqreturn_t (*thread)(int irq, void *p)`: A function which will be used as pollfunc bottom half. This function runs in the context of a kernel thread, and all the processing takes 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 using the `iio_push_to_buffers_with_timestamp()` function.

See below in bold how looks like the triggered buffer setup of the ADXL345 IIO driver that you will develop in the next chapter:

```
int adxl345_core_probe(struct device *dev, struct regmap *regmap,
                      const char *name)
{
    struct iio_dev *indio_dev;
    struct adxl345_data *data;

    [...]

    /* iio_pollfunc_store_time do pf->timestamp = iio_get_time_ns(); */
    devm_iio_triggered_buffer_setup(dev, indio_dev,
                                   &iio_pollfunc_store_time,
                                   adxl345_trigger_handler, NULL);

    devm_iio_device_register(dev, indio_dev);

    return 0;
}

static irqreturn_t adxl345_trigger_handler(int irq, void *p)
{
    struct iio_poll_func *pf = p;
    struct iio_dev *indio_dev = pf->indio_dev;
    struct adxl345_data *data = iio_priv(indio_dev);

    /* 6 bytes axis + 2 bytes padding + 8 bytes timestamp */
    s16 buf[8];
    int i, ret, j = 0, base = DATA0;
    s16 sample;

    /* Read the channels that have been enabled from user space */
    for_each_set_bit(i, indio_dev->active_scan_mask, indio_dev->masklength) {
        ret = regmap_bulk_read(data->regmap, base + i * sizeof(sample),
                               &sample, sizeof(sample));

        if (ret < 0)
```

```

        goto done;
    buf[j++] = sample;
}

iio_push_to_buffers_with_timestamp(indio_dev, buf, pf->timestamp);

done:
iio_trigger_notify_done(indio_dev->trig);
return IRQ_HANDLED;
}

```

Industrial I/O events

The Industrial I/O subsystem provides support for passing hardware generated events up to user space. In IIO, events are not used for passing normal readings from the sensing devices to user space, but rather for out of band information. Normal data reaches user space through a low overhead character device - typically via either software or hardware buffer. The stream format is pseudo fixed, so is described and controlled via sysfs rather than adding headers to the data describing what is in it.

Pretty much, all IIO events correspond to thresholds on some value derived from one or more raw readings from the sensor. They are provided by the underlying hardware. Events have timestamps. Examples include:

- Straight crossing a voltage threshold.
- Moving average crosses a threshold.
- Motion detectors (lots of ways of doing this).
- Thresholds on sum squared or rms values.
- Rate of change thresholds.
- Lots more variants...

The **event sysfs attributes** exposed to user space are specified in the form of bitmasks. Each channel event is specified with an `iio_event_spec` structure:

```

struct iio_event_spec {
    enum iio_event_type type;
    enum iio_event_direction dir;
    unsigned long mask_separate;
    unsigned long mask_shared_by_type;
    unsigned long mask_shared_by_dir;
    unsigned long mask_shared_by_all;
};

```

Where:

- type: Type of the event.
- dir: Direction of the event.

- `mask_separate`: Bit mask of enum `iio_event_info` values; attributes set in this mask will be registered per channel.
- `mask_shared_by_type`: Bit mask of enum `iio_event_info` values; attributes set in this mask will be shared by channel type.
- `mask_shared_by_dir`: Bit mask of enum `iio_event_info` values; attributes set in this mask will be shared by channel type and direction.
- `mask_shared_by_all`: Bit mask of enum `iio_event_info` values; attributes set in this mask will be shared by all channels.

See below the initialization of the `iio_event_spec` structure for the ADXL345 IIO driver that you will develop in the next chapter:

```
static const struct iio_event_spec adxl345_event = {
    .type = IIO_EV_TYPE_THRESH,
    .dir = IIO_EV_DIR_EITHER,
    .mask_separate = BIT(IIO_EV_INFO_VALUE) |
                    BIT(IIO_EV_INFO_PERIOD)
};
```

The `adxl345_event` structure will be integrated in each `iio_chan_spec` structure, as you can see in the following line of code in bold:

```
static const struct iio_chan_spec adxl345_channels[] = {
    ADXL345_CHANNEL(DATA_X0, X, 0),
    ADXL345_CHANNEL(DATA_Y0, Y, 1),
    ADXL345_CHANNEL(DATA_Z0, Z, 2),
    IIO_CHAN_SOFT_TIMESTAMP(3),
};

#define ADXL345_CHANNEL(reg, axis, idx) {
    .type = IIO_ACCEL,
    .modified = 1,
    .channel2 = IIO_MOD_##axis,
    .address = reg,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE) |
                                BIT(IIO_CHAN_INFO_SAMP_FREQ),
    .scan_index = idx,
    .scan_type = {
        .sign = 's',
        .realbits = 13,
        .storagebits = 16,
        .endianness = IIO_LE,
    },
    .event_spec = &adxl345_event,
    .num_event_specs = 1
}
```

You will create the kernel hooks for the user space interactions with the event sysfs attributes:

```
static const struct iio_info adxl345_info = {
    .driver_module      = THIS_MODULE,
    .read_raw           = adxl345_read_raw,
    .write_raw          = adxl345_write_raw,
    .read_event_value    = adxl345_read_event,
    .write_event_value   = adxl345_write_event,
};
```

- `read_event_value`: Read a configuration value associated with the event.
- `write_event_value`: Write a configuration value for the event.

The event notification can be enabled by writing to the sysfs attributes under the `/sys/bus/iio/devices/iio:deviceX/events/` directory.

Delivering IIO events to user space

The `iio_push_event()` function tries to add an event to the list for user space reading. It is usually called within a threaded IRQ.

```
int iio_push_event(struct iio_dev *indio_dev, u64 ev_code, s64 timestamp);
```

- `indio_dev`: Pointer to the IIO device structure.
- `ev_code`: Contains channel type, modifier, direction and event type; these are some macros for packing/unpacking event codes: `IIO_MOD_EVENT_CODE` and `IIO_EVENT_CODE_EXTRACT`.
- `timestamp`: When the event occurred.

In the following code snippet, you will show how to deliver IIO events to user space. You will send these events inside the interrupt handler of the ADXL345 IIO driver that you will develop in the next chapter.

```
/* Interrupt service routine */
static irqreturn_t adxl345_event_handler(int irq, void *handle)
{
    u32 tap_stat, int_stat, int_ret;
    struct iio_dev *indio_dev = handle;
    struct adxl345_data *data = iio_priv(indio_dev);
    data->timestamp = iio_get_time_ns(indio_dev);

    /*
     * ACT_TAP_STATUS should be read before clearing the interrupt
     * Avoid reading ACT_TAP_STATUS in case TAP detection is disabled
     * Read the ACT_TAP_STATUS if any of the axis has been enabled
     */
    if (data->tap_axis_control & (TAP_X_EN | TAP_Y_EN | TAP_Z_EN)) {
        ret = regmap_read(data->regmap, ACT_TAP_STATUS, &tap_stat);
        if (ret) {
            dev_err(data->dev, "error reading ACT_TAP_STATUS register\n");
        }
    }
}
```

```

        return ret;
    }
}
else
    tap_stat = 0;

/*
 * Read the INT_SOURCE (0x30) register.
 * The tap interrupt is cleared
 */
ret = regmap_read(data->regmap, INT_SOURCE, &int_stat);
if (ret) {
    dev_err(data->dev, "error reading INT_SOURCE register\n");
    return ret;
}

/*
 * If the SINGLE_TAP event has occurred, the axl345_do_tap function
 * is called with the ACT_TAP_STATUS register as an argument
 */
if (int_stat & (SINGLE_TAP)) {
    dev_info(data->dev, "single tap interrupt has occurred\n");

    if (tap_stat & TAP_X_EN) {
        iio_push_event(indio_dev,
                       IIO_MOD_EVENT_CODE(IIO_ACCEL,
                                           0,
                                           IIO_MOD_X,
                                           IIO_EV_TYPE_THRESH,
                                           0),
                       data->timestamp);
    }
    if (tap_stat & TAP_Y_EN) {
        iio_push_event(indio_dev,
                       IIO_MOD_EVENT_CODE(IIO_ACCEL,
                                           0,
                                           IIO_MOD_Y,
                                           IIO_EV_TYPE_THRESH,
                                           0),
                       data->timestamp);
    }
    if (tap_stat & TAP_Z_EN) {
        iio_push_event(indio_dev,
                       IIO_MOD_EVENT_CODE(IIO_ACCEL,
                                           0,
                                           IIO_MOD_Z,
                                           IIO_EV_TYPE_THRESH,
                                           0),
                       data->timestamp);
    }
}

return IRQ_HANDLED;
}

```

IIO utils

There are some useful tools that you can use during the development of your IIO driver. They are available under `/tools/iio/` folder in the kernel source tree:

- **lsiio**: Enumerates IIO triggers, devices and accessible channels.
- **iio_event_monitor**: Monitors on IIO device's `ioctl` interface for IIO events.
- **iio_generic_buffer**: Monitors, processes and print data received from an IIO device's buffer.
- **libiio**: A powerful library developed by Analog devices to interface IIO devices. It is available at <https://github.com/analogdevicesinc/libiio>.

LAB 11.1: "IIO Mixed-Signal I/O Device" module

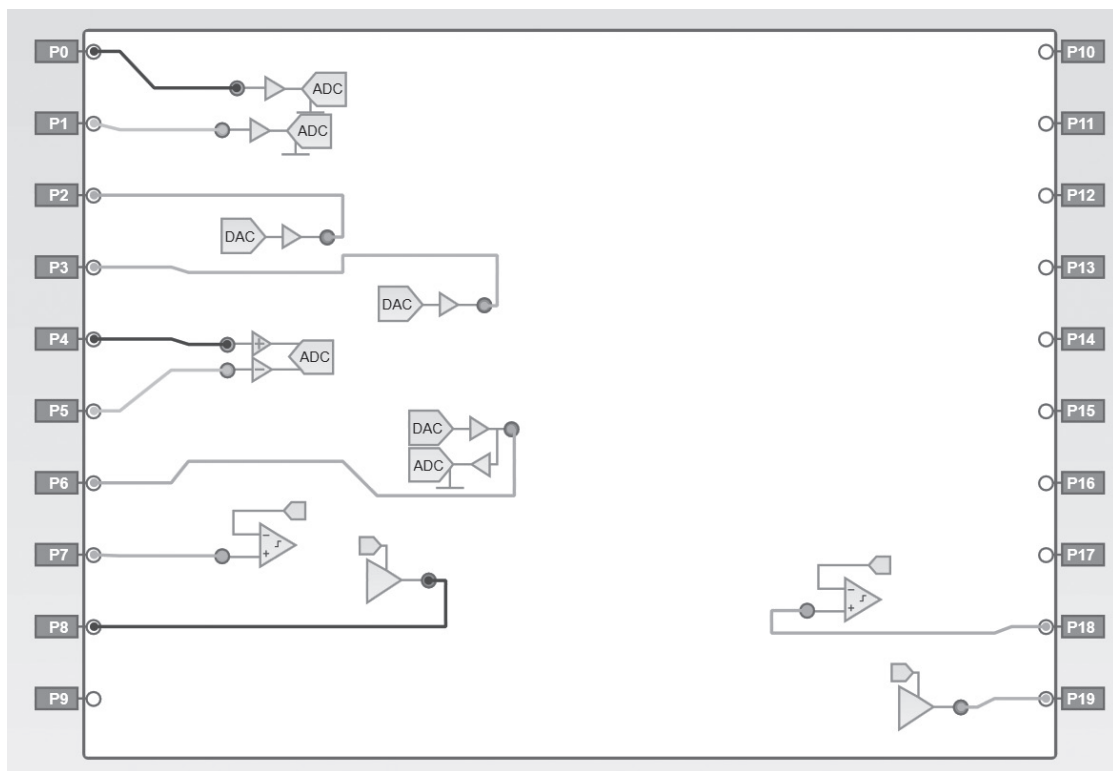
In this lab, you will create an IIO driver to control the Maxim MAX11300 device. You will also develop several user applications to control GPIOs from user space.

The MAX11300 integrates a PIXI™, 12-bit, multichannel, analog-to-digital converter (ADC) and a 12-bit, multichannel, buffered digital-to-analog converter (DAC) in a single integrated circuit (IC). This device offers 20 mixed-signal high-voltage, bipolar ports, which are configurable as an ADC analog input, a DAC analog output, a general-purpose input port (GPI), a general-purpose output port (GPO), or an analog switch terminal. You can check all the info related to this device at <https://www.maximintegrated.com/en/products/analog/data-converters/analog-to-digital-converters/MAX11300.html>.

In this lab, you will use the PIXI™ CLICK board from MIKROE. The documentation of this board can be found at <https://www.mikroe.com/pixi-click>.

Before developing the driver, you can first create a custom design using a GUI software that configures the MAX11300 device. This tool is available for download at the Maxim's website. The MAX11300ConfigurationSetupV1.4.zip tool and the custom design used as a starting point for the development of the driver are included in the lab folder.

In the next screenshot of the tool, you can see the configuration that will be used for the development of the driver:



These are the parameters used during the configuration of the PIXI ports:

- **Port 0 (P0)** -> Single Ended ADC, Average of samples = 1, Reference Voltage = internal, Voltage Range = 0V to 10V
- **Port 1 (P1)** -> Single Ended ADC, Average of samples = 1, Reference Voltage = internal, Voltage Range = 0V to 10V.
- **Port 2 (P2)** -> DAC, Voltage Output Level = 0V, Voltage Range = 0V to 10V.
- **Port 3 (P3)** -> DAC, Voltage Output Level = 0V, Voltage Range = 0V to 10V.
- **Port 4 (P4) and Port 5 (P5)** -> Differential ADC, Pin info: Input Pin (-) is P5 and Input Pin (+) is P4, Reference Voltage = internal, Voltage Range = 0V to 10V.
- **Port 6 (P6)** -> DAC with ADC monitoring, Reference Voltage = internal, Voltage Output Level = 0V, Voltage Range = 0V to 10V.

- **Port 7 (P7)** -> GPI, Interrupt: Masked, Voltage Input Threshold: 2.5V.
- **Port 8 (P8)** -> GPO, Voltage output Level = 3.3V.
- **Port 18 (P18)** -> GPI, Interrupt: Masked, Voltage Input Threshold: 2.5V.
- **Port 19 (P19)** -> GPO, Voltage output Level = 3.3V.

And these are the general parameters used during the configuration of the MAX11300 device:

General Parameter Configuration			
Voltage			
AVSSIO	0 V	AVDDIO	5 V
DVDD	3.3 V	AVDD	5 V
DAC			
☐ Int Voltage Ref	2.5 V	Update Mode	Sequential
Preset Data #1	0 V	Preset Data #2	0 V
ADC			
Int Voltage Ref	2.5 V	Conversion Mode	Continuous Sweep
		Conversion Rate	200 Ksps
Interrupt Mask			
☒ ADC Flag	☒ ADC Data Ready	☒ GPI Data Ready	☒ GPI Data Missed
☒ ADC Data Missed	☒ Voltage Monitor	☒ DAC Driver Over Current	
General			
☐ Soft Reset Control	☐ Sleep Mode		
Serial Interface Burst Mode	Default Address Incrementing Mode		
Configure		Cancel	

Not all the specifications of the MAX11300 device will be covered during the development of this driver. These are the main specifications that will be included:

- Functional modes for ports: Mode 1, Mode 3, Mode 5, Mode 6, Mode 7, Mode 8, Mode 9.
- DAC Update Mode: Sequential.
- ADC Conversion Mode: Continuous Sweep.
- Default ADC Conversion Rate of 200Ksps.
- Interrupts are masked.

LAB 11.1 hardware description

In this lab, you will use the SPI pins of the Raspberry Pi 40-pin GPIO header, which is found on all current Raspberry Pi boards, to connect to the PIXI™ CLICK mikroBUS™ socket. In the following image, you can see the PIXI™ CLICK mikroBUS™ socket:

Notes	Pin					Pin	Notes
	NC	1	AN	PWM	16	CNV	ADC trigger control
	NC	2	RST	INT	15	INT	Interrupt output
Chip select	CS	3	CS	RX	14	NC	
SPI clock	SCK	4	SCK	TX	13	NC	
SPI data output	SDO	5	MISO	SCL	12	NC	
SPI data input	SDI	6	MOSI	SDA	11	NC	
Power supply	+3.3V	7	3.3V	5V	10	+5V	Power supply
Ground	GND	8	GND	GND	9	GND	Ground

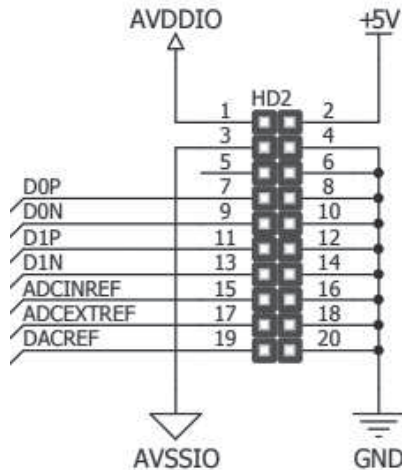
Connect the Raspberry Pi’s SPI pins to the SPI ones of the MAX11300 device:

- Connect Raspberry Pi **GPIO8 (CE0)** to MAX11300 **CS** (Pin 3 of Mikrobus)
- Connect Raspberry Pi **SCLK** to MAX11300 **SCK** (Pin 4 of Mikrobus)
- Connect Raspberry Pi **MOSI** to MAX11300 **MOSI** (Pin 6 of Mikrobus)
- Connect Raspberry Pi **MISO** to MAX11300 **MISO** (Pin 5 of Mikrobus)

You will also connect the next power pins between the two boards:

- Connect Raspberry Pi **3.3V** to MAX11300 **3.3V** (Pin 7 of Mikrobus)
- Connect Raspberry Pi **5V** to MAX11300 **5V** (Pin 10 of Mikrobus)
- Connect Raspberry Pi **GNDs** to MAX11300 **GNDs** (Pin 9 and Pin 8 of Mikrobus)

Finally, find the HD2 connector in the schematics of the PIXI™ CLICK board located at <https://download.mikroe.com/documents/add-on-boards/click/pixi/pixi-click-schematic-v100.pdf>. In the following image, the connector is also shown:



In the HD2 connector, connect the following pins:

- Connect the Pin 2 of HD2 (+5V) to the Pin 1 of HD2 (AVDDIO)
- Connect the Pin 4 of HD2 (GND) to the Pin 3 of HD2 (AVSSIO)

The hardware setup between the two boards is already done!!

LAB 11.1 Device Tree description

Open the `bcm2710-rpi-3-b.dts` DT file and find the `spi0` controller master node. In the `spi0` node, you can see a `pinctrl-0` property which points to the pin configuration nodes (`spi0_pins` and `spi0_cs_pins`) that configure the pins of the `spi0` controller in SPI mode.

The `cs-gpios` property specifies the `gpio` pins to be used for chip selects. In the `spi0` node, you can see that there are two chip selects enabled. You will only use the first chip select `<gpio 8 1>` during the development of this lab. Comment out all the sub-nodes included in the `spi0` node that are coming from previous labs to avoid the errors during the compilation of the Device Tree.

Now, you will add the `max11300` node to the `spi0` controller node. The `max11300` node includes twenty sub-nodes which represent the different ports of the MAX11300 device. The first two properties inside the `max11300` node are `#size-cells` and `#address-cells`. The `#address-cells` property defines the number of `<u32>` cells used to encode the address field in the child node's `reg` property. The `#size-cells` property defines the number of `<u32>` cells used to encode the size field in the child node's `reg` property. In this driver, the `#address-cells` property of the `max11300` node is set to 1 and the `#size-cells` property is set to 0. These settings specify the following: one cell is required to represent an address, and there is no a required cell to represent the size of the child nodes of the `max11300` node. The `reg` property included in all the channel sub-nodes follows the specification that was set in the `#address-cells` property of the parent `max11300` node.

There must be a DT device node's `compatible` property identical to the `compatible` string stored in one of the driver's `of_device_id` structures.

The `spi-max-frequency` specifies the maximum SPI clocking speed of the device in Hz.

Each of the twenty child nodes can include the following properties:

- **reg** -- This property sets the port number of the MAX11300 device.
- **port-mode** -- This property sets the port configuration for the selected port.
- **AVR** -- This property selects the ADC voltage reference: 0: Internal, 1: External.
- **adc-range** -- This property selects the voltage range for ADC related modes.
- **dac-range** -- This property selects the voltage range for DAC related modes.
- **adc-samples** -- This property selects the number of samples for ADC related modes.
- **negative-input** -- This property sets the negative port number for ports configured in mode 8.

The channel sub-nodes will be configured with the same parameters that were used during the configuration of the MAX11300 GUI software:

```

&spi0 {
    pinctrl-names = "default";
    pinctrl-0 = <&spi0_pins &spi0_cs_pins>;
    cs-gpios = <&gpio 8 1>, <&gpio 7 1>;

    /* CE0 */
    /*spidev0: spidev@0{
        compatible = "spidev";
        reg = <0>;
        #address-cells = <1>;
        #size-cells = <0>;
        spi-max-frequency = <125000000>;
    };*/

    /* CE1 */
    /*spidev1: spidev@1{
        compatible = "spidev";
        reg = <1>;
        #address-cells = <1>;
        #size-cells = <0>;
        spi-max-frequency = <125000000>;
    };*/

    max11300@0 {
        #size-cells = <0>;
        #address-cells = <1>;
        compatible = "maxim,max11300";
        reg = <0>;

        spi-max-frequency = <10000000>;

        channel@0 {
            reg = <0>;
            port-mode = <PORT_MODE_7>;
            AVR = <0>;
            adc-range = <ADC_VOLTAGE_RANGE_PLUS10>;
            adc-samples = <ADC_SAMPLES_1>;
        };
        channel@1 {
            reg = <1>;
            port-mode = <PORT_MODE_7>;
            AVR = <0>;
            adc-range = <ADC_VOLTAGE_RANGE_PLUS10>;
            adc-samples = <ADC_SAMPLES_128>;
        };
        channel@2 {
            reg = <2>;
            port-mode = <PORT_MODE_5>;
            dac-range = <DAC_VOLTAGE_RANGE_PLUS10>;
        };
        channel@3 {
            reg = <3>;
            port-mode = <PORT_MODE_5>;
            dac-range = <DAC_VOLTAGE_RANGE_PLUS10>;
        };
    };
}

```

```

};
channel@4 {
    reg = <4>;
    port-mode = <PORT_MODE_8>;
    AVR = <0>;
    adc-range = <ADC_VOLTAGE_RANGE_PLUS10>;
    adc-samples = <ADC_SAMPLES_1>;
    negative-input = <5>;
};
channel@5 {
    reg = <5>;
    port-mode = <PORT_MODE_9>;
    AVR = <0>;
    adc-range = <ADC_VOLTAGE_RANGE_PLUS10>;
};
channel@6 {
    reg = <6>;
    port-mode = <PORT_MODE_6>;
    AVR = <0>;
    dac-range = <DAC_VOLTAGE_RANGE_PLUS10>;
};
channel@7 {
    reg = <7>;
    port-mode = <PORT_MODE_1>;
};
channel@8 {
    reg = <8>;
    port-mode = <PORT_MODE_3>;
};
channel@9 {
    reg = <9>;
    port-mode = <PORT_MODE_0>;
};
channel@10 {
    reg = <10>;
    port-mode = <PORT_MODE_0>;
};
channel@11 {
    reg = <11>;
    port-mode = <PORT_MODE_0>;
};
channel@12 {
    reg = <12>;
    port-mode = <PORT_MODE_0>;
};
channel@13 {
    reg = <13>;
    port-mode = <PORT_MODE_0>;
};
channel@14 {
    reg = <14>;
    port-mode = <PORT_MODE_0>;
};
channel@15 {

```

```

        reg = <15>;
        port-mode = <PORT_MODE_0>;
    };
    channel@16 {
        reg = <16>;
        port-mode = <PORT_MODE_0>;
    };
    channel@17 {
        reg = <17>;
        port-mode = <PORT_MODE_0>;
    };
    channel@18 {
        reg = <18>;
        port-mode = <PORT_MODE_1>;
    };
    channel@19 {
        reg = <19>;
        port-mode = <PORT_MODE_3>;
    };
};

/*Accel: ADXL345@0 {
    compatible = "arrow,adx1345";
    spi-max-frequency = <5000000>;
    spi-cpol;
    spi-cpha;
    reg = <0>;
    pinctrl-0 = <&accel_int_pin>;
    int-gpios = <&gpio 23 0>;
    interrupts = <23 1>;
    interrupt-parent = <&gpio>;
};*/
};

```

You also have to include the next header file in bold inside the bcm2710-rpi-3-b.dts file:

```

/dts-v1/;

#include "bcm2710.dtsi"
#include "bcm2709-rpi.dtsi"
#include "bcm283x-rpi-smc9514.dtsi"
#include "bcm283x-rpi-cs11-2lane.dtsi"
#include "bcm283x-rpi-i2c0mux_0_44.dtsi"
#include "bcm271x-rpi-bt.dtsi"
#include <dt-bindings/iio/maxim,max11300.h>

```

The maxim,max11300.h file includes the values of the DT binding properties that will be used for the channel sub-nodes. You have to place the maxim,max11300.h file under the next iio folder in the kernel source tree:

```
~/linux_rpi3/linux/include/dt-bindings/iio/
```

The content of the maxim,max11300.h file is shown below:

```

#ifndef _DT_BINDINGS_MAXIM_MAX11300_H
#define _DT_BINDINGS_MAXIM_MAX11300_H

#define PORT_MODE_0 0
#define PORT_MODE_1 1
#define PORT_MODE_2 2
#define PORT_MODE_3 3
#define PORT_MODE_4 4
#define PORT_MODE_5 5
#define PORT_MODE_6 6
#define PORT_MODE_7 7
#define PORT_MODE_8 8
#define PORT_MODE_9 9
#define PORT_MODE_10 10
#define PORT_MODE_11 11
#define PORT_MODE_12 12

#define ADC_SAMPLES_1 0
#define ADC_SAMPLES_2 1
#define ADC_SAMPLES_4 2
#define ADC_SAMPLES_8 3
#define ADC_SAMPLES_16 4
#define ADC_SAMPLES_32 5
#define ADC_SAMPLES_64 6
#define ADC_SAMPLES_128 7

/* ADC voltage ranges */
#define ADC_VOLTAGE_RANGE_NOT_SELECTED 0
#define ADC_VOLTAGE_RANGE_PLUS10 1 // 0 to +5V range
#define ADC_VOLTAGE_RANGE_PLUSMINUS5 2 // -5V to +5V range
#define ADC_VOLTAGE_RANGE_MINUS10 3 // -10V to 0 range
#define ADC_VOLTAGE_RANGE_PLUS25 4 // 0 to +2.5 range

/* DAC voltage ranges mode 5 */
#define DAC_VOLTAGE_RANGE_NOT_SELECTED 0
#define DAC_VOLTAGE_RANGE_PLUS10 1
#define DAC_VOLTAGE_RANGE_PLUSMINUS5 2
#define DAC_VOLTAGE_RANGE_MINUS10 3

#endif /* _DT_BINDINGS_MAXIM_MAX11300_H */

```

LAB 11.1 driver description

The main code sections of the driver will be described using three different categories: Industrial framework as an SPI interaction, Industrial framework as an IIO device and GPIO driver interface. The MAX11300 driver is based on the Paul Cercueil's AD5592R driver:

<https://elixir.bootlin.com/linux/latest/source/drivers/iio/dac/ad5592r.c>

Industrial framework as an SPI interaction

These are the main code sections:

1. Include the required header files:

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

2. Create a spi_driver structure:

```
static struct spi_driver max11300_spi_driver = {
    .driver = {
        .name = "max11300",
        .of_match_table = of_match_ptr(max11300_of_match),
    },
    .probe = max11300_spi_probe,
    .remove = max11300_spi_remove,
    .id_table = max11300_spi_ids,
};

module_spi_driver(max11300_spi_driver);
```

3. Register to the SPI bus as a driver:

```
module_spi_driver(max11300_spi_driver);
```

4. Add "maxim,max11300" to the list of devices supported by the driver. The compatible variable matches with the compatible property of the max11300 DT node:

```
static const struct of_device_id max11300_of_match[] = {
    { .compatible = "maxim,max11300", },
    {},
};

MODULE_DEVICE_TABLE(of, max11300_of_match);
```

5. Define an array of spi_device_id structures:

```
static const struct spi_device_id max11300_spi_ids[] = {
    { .name = "max11300", },
    {}
};

MODULE_DEVICE_TABLE(spi, max11300_spi_ids);
```

6. Initialize the max11300_rw_ops structure with the read and write functions that will access via SPI to the registers of the MAX11300 device. See below the code of these functions:

```

/*
 * Initialize the struct max11300_rw_ops with callback functions
 * that will write/read via SPI the MAX11300 registers
 */
static const struct max11300_rw_ops max11300_rw_ops = {
    .reg_write = max11300_reg_write,
    .reg_read = max11300_reg_read,
    .reg_read_differential = max11300_reg_read_differential,
};

/* Function to write the MAX11300 registers */
static int max11300_reg_write(struct max11300_state *st, u8 reg, u16 val)
{
    struct spi_device *spi = container_of(st->dev, struct spi_device, dev);

    struct spi_transfer t[] = {
        {
            .tx_buf = &st->tx_cmd,
            .len = 1,
        }, {
            .tx_buf = &st->tx_msg,
            .len = 2,
        },
    };

    /* To transmit via SPI, the LSB bit of the command byte must be 0 */
    st->tx_cmd = (reg << 1);

    /*
     * In little endian CPUs, the byte stored in the higher address of the
     * "val" variable (MSB of the DAC) will be stored in the lower address of the
     * "st->tx_msg" variable using cpu_to_be16()
     */
    st->tx_msg = cpu_to_be16(val);

    return spi_sync_transfer(spi, t, ARRAY_SIZE(t));
}

/* Function to read the MAX11300 registers in SE mode */
static int max11300_reg_read(struct max11300_state *st, u8 reg, u16 *value)
{
    struct spi_device *spi = container_of(st->dev, struct spi_device, dev);
    int ret;

    struct spi_transfer t[] = {
        {
            .tx_buf = &st->tx_cmd,
            .len = 1,
        }, {
            .rx_buf = &st->rx_msg,
            .len = 2,
        },
    };

    dev_info(st->dev, "read SE channel\n");

```

```

/* To receive via SPI the LSB bit of the command byte must be 1 */
st->tx_cmd = ((reg << 1) | 1);

ret = spi_sync_transfer(spi, t, ARRAY_SIZE(t));
if (ret < 0)
    return ret;

/*
 * In little endian CPUs the first byte (MSB of the ADC) received via
 * SPI (in BE format) will be stored in the lower address of "st->rx_msg"
 * variable. This byte is copied to the higher address of the "value"
 * variable using be16_to_cpu(). The second byte received via SPI is
 * copied from the higher address of "st->rx_msg" to the lower address
 * of the "value" variable in little endian CPUs.
 * In big endian CPUs the addresses are not swapped
 */
*value = be16_to_cpu(st->rx_msg);

return 0;
}
/* Function to read the MAX11300 registers in differential mode (2's complement) */
static int max11300_reg_read_differential(struct max11300_state *st, u8 reg, int *value)
{
    struct spi_device *spi = container_of(st->dev, struct spi_device, dev);
    int ret;

    struct spi_transfer t[] = {
        {
            .tx_buf = &st->tx_cmd,
            .len = 1,
        }, {
            .rx_buf = &st->rx_msg,
            .len = 2,
        },
    };

    dev_info(st->dev, "read differential channel\n");

    /* LSB of command byte has to be 1 */
    st->tx_cmd = ((reg << 1) | 1);

    ret = spi_sync_transfer(spi, t, ARRAY_SIZE(t));
    if (ret < 0)
        return ret;

    /*
     * Extend to an int 2's complement value the received SPI value in 2's
     * complement value, which is stored in the "st->rx_msg" variable
     */
    *value = sign_extend32(be16_to_cpu(st->rx_msg), 11);

    return 0;
}

```


Industrial framework as an IIO device

These are the main code sections:

1. Include the required header files:

```
#include <linux/iio/iio.h> /* devm_iio_device_alloc(), iio_priv() */
```

2. Create a global private data structure to manage the device from any function of the driver:

```
struct max11300_state {
    struct device *dev; // pointer to SPI device
    const struct max11300_rw_ops *ops; // pointer to spi callback functions
    struct gpio_chip gpiochip; // gpio_chip controller
    struct mutex gpio_lock;
    u8 num_ports; // number of ports of the MAX11300 device = 20
    u8 num_gpios; // number of ports declared in the DT as GPIOs
    u8 gpio_offset[20]; // gpio port numbers (0 to 19) for the "offset" values in the
range 0..(@ngpio - 1)
    u8 gpio_offset_mode[20]; // gpio port modes (1 and 3) for the "offset" values in
the range 0..(@ngpio - 1)
    u8 port_modes[20]; // port modes for the 20 ports of the MAX11300
    u8 adc_range[20]; // voltage range for ADC related modes
    u8 dac_range[20]; // voltage range for DAC related modes
    u8 adc_reference[20]; // ADC voltage reference: 0: Internal, 1: External
    u8 adc_samples[20]; // number of samples for ADC related modes
    u8 adc_negative_port[20]; // negative port number for ports configured in mode 8
    u8 tx_cmd; // command byte for SPI transactions
    __be16 tx_msg; // transmit value for SPI transactions in BE format
    __be16 rx_msg; // value received in SPI transactions in BE format
};
```

3. In the max11300_probe() function, declare an instance of the private structure, and allocate the iio_dev structure:

```
struct iio_dev *indio_dev;
struct max11300_state *st;
indio_dev = devm_iio_device_alloc(dev, sizeof(*st));
```

4. Initialize the iio_device and the data private structure within the max11300_probe() function. The data private structure will be previously allocated by using the iio_priv() function. Keep pointers between physical devices (devices as handled by the physical bus, SPI in this case) and logical devices:

```
st = iio_priv(indio_dev); /* To be able to access the private data structure in other
parts of the driver, you need to attach it to the iio_dev structure using the iio_priv()
function. You will retrieve the pointer to the private structure using the same function
iio_priv() */

st->dev = dev; /* Store pointer to the SPI device, needed for exchanging data with the
MAX11300 device */
```

```

dev_set_drvdata(dev, iio_dev); /* Link the spi device with the iio device */

iio_dev->name = name; /* Store the iio_dev name. Before doing this within your probe()
function, you will get the spi_device_id that triggered the match using spi_get_device_
id() */

iio_dev->dev.parent = dev; /* Keep pointers between physical devices (devices as handled
by the physical bus, SPI in this case) and logical devices */

indio_dev->info = &max11300_info; /* Store the address of the iio_info structure, which
contains a pointer variable to the IIO raw reading/writing callbacks */

max11300_alloc_ports(st); /* Configure the IIO channels of the device to generate the IIO
sysfs entries. This function will be described in more detail in the next point */

```

5. The `max11300_alloc_ports()` function reads the DT properties from the channel sub-nodes of the `max11300` node by using the `fwnode_property_read_u32()` function, and it stores the values of these properties into the variables of the data global structure. The `max11300_set_port_modes()` function uses these variables to configure the ports of the MAX11300 device. The `max11300_alloc_ports()` function generates the different IIO sysfs entries by using the `max11300_setup_port_*_mode()` functions:

```

/*
 * This function will allocate and configure the iio channels of the iio device.
 * It will also read the DT properties of each port (channel) and will store
 * them in the global structure of the device
 */
static int max11300_alloc_ports(struct max11300_state *st)
{
    unsigned int i, curr_port = 0, num_ports = st->num_ports, port_mode_6_count = 0,
    offset = 0;
    st->num_gpios = 0;

    /* Recover the iio device from the global structure */
    struct iio_dev *iio_dev = iio_priv_to_dev(st);

    /* Pointer to the storage of the specs of all the iio channels */
    struct iio_chan_spec *ports;

    /* Pointer to struct fwnode_handle, allowing device description object */
    struct fwnode_handle *child;

    u32 reg, tmp;
    int ret;

    /*
     * Walks for each MAX11300 child node in the DT;
     * if an error is found in the node, then walks to
     * the following one (continue)
     */
    device_for_each_child_node(st->dev, child) {
        ret = fwnode_property_read_u32(child, "reg", &reg);
        if (ret || reg >= ARRAY_SIZE(st->port_modes))

```

```

        continue;

/*
 * Store the value of the DT "port,mode" property
 * in the global structure to know the mode of each port in
 * other functions of the driver
 */
ret = fwnode_property_read_u32(child, "port-mode", &tmp);
if (!ret)
    st->port_modes[reg] = tmp;

/* All the DT nodes should include the port-mode property */
else {
    dev_info(st->dev, "port mode is not found\n");
    continue;
}

/*
 * You will store other DT properties
 * depending of the used "port,mode" property
 */
switch (st->port_modes[reg]) {
case PORT_MODE_7:
    ret = fwnode_property_read_u32(child, "adc-range", &tmp);
    if (!ret)
        st->adc_range[reg] = tmp;
    else
        dev_info(st->dev, "Get default ADC range\n");

    ret = fwnode_property_read_u32(child, "AVR", &tmp);
    if (!ret)
        st->adc_reference[reg] = tmp;
    else
        dev_info(st->dev, "Get default internal ADC reference\n");

    ret = fwnode_property_read_u32(child, "adc-samples", &tmp);
    if (!ret)
        st->adc_samples[reg] = tmp;
    else
        dev_info(st->dev, "Get default internal ADC sampling\n");

    break;

case PORT_MODE_8:
    ret = fwnode_property_read_u32(child, "adc-range", &tmp);
    if (!ret)
        st->adc_range[reg] = tmp;
    else
        dev_info(st->dev, "Get default ADC range\n");

    ret = fwnode_property_read_u32(child, "AVR", &tmp);
    if (!ret)
        st->adc_reference[reg] = tmp;
    else

```

```

        dev_info(st->dev, "Get default internal ADC reference\n");

ret = fwnode_property_read_u32(child, "adc-samples", &tmp);
if (!ret)
    st->adc_samples[reg] = tmp;
else
    dev_info(st->dev, "Get default internal ADC sampling\n");

ret = fwnode_property_read_u32(child, "negative-input", &tmp);
if (!ret)
    st->adc_negative_port[reg] = tmp;
else {
    dev_info(st->dev, "Bad value for negative ADC channel\n");
    return -EINVAL;
}

break;

case PORT_MODE_9: case PORT_MODE_10:
    ret = fwnode_property_read_u32(child, "adc-range", &tmp);
    if (!ret)
        st->adc_range[reg] = tmp;
    else
        dev_info(st->dev, "Get default ADC range\n");

    ret = fwnode_property_read_u32(child, "AVR", &tmp);
    if (!ret)
        st->adc_reference[reg] = tmp;
    else
        dev_info(st->dev, "Get default internal ADC reference\n");

    break;

case PORT_MODE_5: case PORT_MODE_6:
    ret = fwnode_property_read_u32(child, "dac-range", &tmp);
    if (!ret)
        st->dac_range[reg] = tmp;
    else
        dev_info(st->dev, "Get default DAC range\n");

/*
 * A port in mode 6 will generate two IIO sysfs entries,
 * one for writing the DAC port, and another for reading
 * the ADC port
 */
if ((st->port_modes[reg]) == PORT_MODE_6) {
    ret = fwnode_property_read_u32(child, "AVR", &tmp);
    if (!ret)
        st->adc_reference[reg] = tmp;
    else
        dev_info(st->dev, "Get default internal
            ADC reference\n");

/*

```

```

        * Get the number of ports set in mode_6 to
        * allocate space for the realated iio channels
        */
        port_mode_6_count++;
    }

    break;

/* The port is configured as a GPI in the DT */
case PORT_MODE_1:
    /*
     * Link the gpio offset with the port number,
     * starting with offset = 0
     */
    st->gpio_offset[offset] = reg;

    /*
     * Store the port_mode for each gpio offset,
     * starting with offset = 0
     */
    st->gpio_offset_mode[offset] = PORT_MODE_1;

    /*
     * Increment the gpio offset and number of configured
     * ports as GPIOs
     */
    offset++;
    st->num_gpios++;
    break;

/* The port is configured as a GPO in the DT */
case PORT_MODE_3:
    /*
     * Link the gpio offset with the port number,
     * starting with offset = 0
     */
    st->gpio_offset[offset] = reg;

    /*
     * Store the port_mode for each gpio offset,
     * starting with offset = 0
     */
    st->gpio_offset_mode[offset] = PORT_MODE_3;

    /*
     * Increment the gpio offset and
     * number of configured ports as GPIOs
     */
    offset++;
    st->num_gpios++;
    break;

case PORT_MODE_0:
    dev_info(st->dev, "the channel %d is set in default port

```

```

                                mode_0\n", reg);
                                break;

                                default:
                                dev_info(st->dev, "bad port mode for channel %d\n", reg);
                                }

                                }

                                /*
                                * Allocate space for the storage of all the IIO channels specs.
                                * Returns a pointer to this storage
                                */
                                devm_kcalloc(st->dev, num_ports + port_mode_6_count, sizeof(*ports), GFP_KERNEL);

                                /*
                                * i is the number of the channel, &ports[curr_port] is a pointer
                                * variable that will store the "iio_chan_spec structure" address of
                                * each port
                                */
                                for (i = 0; i < num_ports; i++) {
                                    switch (st->port_modes[i]) {
                                        case PORT_MODE_5:
                                            max11300_setup_port_5_mode(iio_dev, &ports[curr_port],
                                                                 true, i, PORT_MODE_5);

                                            curr_port++;
                                            break;

                                        case PORT_MODE_6:
                                            max11300_setup_port_6_mode(iio_dev, &ports[curr_port],
                                                                 true, i, PORT_MODE_6);

                                            curr_port++;
                                            max11300_setup_port_6_mode(iio_dev, &ports[curr_port],
                                                                 false, i, PORT_MODE_6);

                                            curr_port++;
                                            break;

                                        case PORT_MODE_7:
                                            max11300_setup_port_7_mode(iio_dev, &ports[curr_port],
                                                                 false, i, PORT_MODE_7);

                                            curr_port++;
                                            break;

                                        case PORT_MODE_8:
                                            max11300_setup_port_8_mode(iio_dev, &ports[curr_port],
                                                                 false, i, st->adc_negative_port[i],
                                                                 PORT_MODE_8);

                                            curr_port++;
                                            break;

                                        case PORT_MODE_0:
                                            dev_info(st->dev, "the channel is set in default port mode_0\n");
                                            break;

```

```

        case PORT_MODE_1:
            dev_info(st->dev, "the channel %d is set in port mode_1\n", i);
            break;

        case PORT_MODE_3:
            dev_info(st->dev, "the channel %d is set in port mode_3\n", i);
            break;

        default:
            dev_info(st->dev, "bad port mode for channel %d\n", i);
    }
}

iio_dev->num_channels = curr_port;
iio_dev->channels = ports;

return 0;
}

```

6. Write the `iio_info` structure. The read/write user space operations to the sysfs data channel access attributes are mapped to the following kernel callbacks:

```

static const struct iio_info max11300_info = {
    .read_raw = max11300_read_adc,
    .write_raw = max11300_write_dac,
};

```

The `max11300_write_dac()` function contains a `switch(mask)` that selects different tasks, depending on the received parameter values. If the received `info_mask` value is `[IIO_CHAN_INFO_RAW] = "raw"`, the `max11300_reg_write()` function is called, which writes a DAC value (using an SPI transaction) to the selected port DAC data register.

When the `max11300_read_adc()` function receives the `info_mask` value `[IIO_CHAN_INFO_RAW] = "raw"`, it first reads the value of the ADC channel address to select the ADC port mode. Once the ADC port mode has been selected, then `max11300_reg_read()` or `max11300_reg_read_differential()` functions are called, which read the value of the selected port ADC data register via an SPI transaction. The returned ADC value is stored into the `val` variable, then this value is returned to user space through the `IIO_VAL_INT` identifier.

GPIO driver interface

The MAX11300 driver will also include a GPIO controller, which configures and controls the MAX11300 ports selected as GPIOs (Port 1 and Port 3 modes) in the DT node of the device.

In the Chapter 5 of this book, you saw how to control GPIOs from kernel space by using the GPIO descriptor consumer interface of the GPIOLib framework.

Most processors today use composite pin controllers. These composite pin controllers will control the GPIOs of the processor, generate interrupts on top of the GPIO functionality and allow pin multiplexing using the I/O pins of the processor as GPIOs or as one of several peripheral functions. The composite pin controllers are configured by using a pinctrl driver.

The pinctrl driver registers the `gpio_chip` structures with the kernel, the `irq_chip` structures with the IRQ system and the `pinctrl_desc` structures with the Pinctrl subsystem. The `gpio` and `pin` controllers are associated with each other within the pinctrl driver through the `pinctrl_add_gpio_range()` function, which adds a range of GPIOs to be handled by a certain pin controller. In the section 2.1 of the `gpio` Device Tree binding document, located at `Documentation/devicetree/bindings/gpio/gpio.txt` in the kernel source tree, you can see the `gpio` and `pin` controller interaction within the DT sources.

The GPIOLib framework provides the kernel and user space APIs to control the GPIOs.

Our MAX11300 IIO driver will include a basic GPIO controller, which configures the ports of the MAX11300 device as GPIOs, sets the direction of the GPIOs (input or output) and controls the output level of the GPIO lines (low or high output level).

These are the main steps to create the GPIO controller in our MAX11300 IIO driver:

1. Include the following header, which declares the structures used to define a GPIO driver:

```
#include <linux/gpio/driver.h>
```

2. Initialize the `gpio_chip` structure with the different callbacks that will control the `gpio` lines of the GPIO controller, and register the `gpio` chip with the kernel by using the `gpiochip_add_data()` function:

```
static int max11300_gpio_init(struct max11300_state *st)
{
    st->gpiochip.label = "gpio-max11300";
    st->gpiochip.base = -1;
    st->gpiochip.ngpio = st->num_gpios;
    st->gpiochip.parent = st->dev;
    st->gpiochip.can_sleep = true;
    st->gpiochip.direction_input = max11300_gpio_direction_input;
    st->gpiochip.direction_output = max11300_gpio_direction_output;
    st->gpiochip.get = max11300_gpio_get;
    st->gpiochip.set = max11300_gpio_set;
```



```

    st->gpiochip.owner = THIS_MODULE;

    /* Register a gpio_chip */
    return gpiochip_add_data(&st->gpiochip, st);
}

```

3. These are the callback functions that will control the GPIO lines of the MAX11300 GPIO controller:

```

/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the GPI_DATA registers of the MAX11300
 */
static int max11300_gpio_get(struct gpio_chip *chip, unsigned int offset)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    int ret = 0;
    u16 read_val;
    u8 reg;
    int val;

    mutex_lock(&st->gpio_lock);

    if (st->gpio_offset_mode[offset] == PORT_MODE_3)
        dev_info(st->dev, "the gpio %d cannot be configured in input mode\n", offset);

    /* for GPIOs from 16 to 19 ports */
    if (st->gpio_offset[offset] > 0x0F) {
        reg = GPI_DATA_19_TO_16_ADDRESS;
        ret = st->ops->reg_read(st, reg, &read_val);
        if (ret)
            goto err_unlock;

        val = (int) (read_val);
        val = val << 16;

        if (val & BIT(st->gpio_offset[offset]))
            val = 1;
        else
            val = 0;

        mutex_unlock(&st->gpio_lock);
        return val;
    }
    else {
        reg = GPI_DATA_15_TO_0_ADDRESS;
        ret = st->ops->reg_read(st, reg, &read_val);
        if (ret)
            goto err_unlock;

        val = (int) read_val;
    }
}

```

```

        if(val & BIT(st->gpio_offset[offset]))
            val = 1;
        else
            val = 0;

        mutex_unlock(&st->gpio_lock);
        return val;
    }

err_unlock:
    mutex_unlock(&st->gpio_lock);
    return ret;
}

/*
 * struct gpio_chip set callback function.
 * It sets the output value of the GPIO line with
 * GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO_DATA registers of the max11300
 */
static void max11300_gpio_set(struct gpio_chip *chip, unsigned int offset, int value)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    u8 reg;
    unsigned int val = 0;

    mutex_lock(&st->gpio_lock);

    if (st->gpio_offset_mode[offset] == PORT_MODE_1)
        dev_info(st->dev, "the gpio %d cannot accept this output\n", offset);

    if (value == 1 && (st->gpio_offset[offset] > 0x0F)) {
        dev_info(st->dev,
            "The GPIO output is set high and port_number is %d. Pin is > 0x0F\n",
            st->gpio_offset[offset]);
        val |= BIT(st->gpio_offset[offset]);
        val = val >> 16;
        reg = GPO_DATA_19_TO_16_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 0 && (st->gpio_offset[offset] > 0x0F)) {
        dev_info(st->dev,
            "The GPIO output is set low and port_number is %d. Pin is > 0x0F\n",
            st->gpio_offset[offset]);
        val &= ~BIT(st->gpio_offset[offset]);
        val = val >> 16;
        reg = GPO_DATA_19_TO_16_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 1 && (st->gpio_offset[offset] < 0x0F)) {
        dev_info(st->dev,
            "The GPIO output is set high and port_number is %d. Pin is < 0x0F\n",
            st->gpio_offset[offset]);
        val |= BIT(st->gpio_offset[offset]);
    }
}

```

```

        reg = GPO_DATA_15_TO_0_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 0 && (st->gpio_offset[offset] < 0x0F)) {
        dev_info(st->dev,
            "The GPIO output is set low and port_number is %d. Pin is < 0x0F\n",
                st->gpio_offset[offset]);
        val &= ~BIT(st->gpio_offset[offset]);
        reg = GPO_DATA_15_TO_0_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else
        dev_info(st->dev, "the gpio %d cannot accept this value\n", offset);

    mutex_unlock(&st->gpio_lock);
}

/*
 * struct gpio_chip direction_input callback function.
 * It configures the GPIO port as an input (GPI),
 * writing to the PORT_CFG register of the max11300
 */
static int max11300_gpio_direction_input(struct gpio_chip *chip, unsigned int offset)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    int ret;
    u8 reg;
    u16 port_mode, val;

    mutex_lock(&st->gpio_lock);

    /* Get the port number stored in the GPIO offset */
    if (st->gpio_offset_mode[offset] == PORT_MODE_3)
        dev_info(st->dev, "Error.The gpio %d only can be set in output mode\n",
            offset);

    /* Set the logic 1 input above 2.5V level */
    val = 0xffff;

    /* Store the GPIO threshold value in the port DAC register */
    reg = PORT_DAC_DATA_BASE_ADDRESS + st->gpio_offset[offset];
    ret = st->ops->reg_write(st, reg, val);
    if (ret)
        goto err_unlock;

    /* Configure the port as GPI */
    reg = PORT_CFG_BASE_ADDRESS + st->gpio_offset[offset];
    port_mode = (1 << 12);
    ret = st->ops->reg_write(st, reg, port_mode);
    if (ret)
        goto err_unlock;

    mdelay(1);

```

```

err_unlock:
    mutex_unlock(&st->gpio_lock);

    return ret;
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO port as an output (GPO) writing to
 * the PORT_CFG register of the max11300, and it sets output value of the
 * GPIO line with GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO data registers of the max11300
 */
static int max11300_gpio_direction_output(struct gpio_chip *chip,
                                         unsigned int offset, int value)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    int ret;
    u8 reg;
    u16 port_mode, val;

    mutex_lock(&st->gpio_lock);

    dev_info(st->dev, "The GPIO is set as an output\n");

    if (st->gpio_offset_mode[offset] == PORT_MODE_1)
        dev_info(st->dev, "the gpio %d only can be set in input mode\n", offset);

    /* GPIO output high is 3.3V */
    val = 0x0547;

    reg = PORT_DAC_DATA_BASE_ADDRESS + st->gpio_offset[offset];
    ret = st->ops->reg_write(st, reg, val);
    if (ret) {
        mutex_unlock(&st->gpio_lock);
        return ret;
    }
    mdelay(1);
    reg = PORT_CFG_BASE_ADDRESS + st->gpio_offset[offset];
    port_mode = (3 << 12);
    ret = st->ops->reg_write(st, reg, port_mode);
    if (ret) {
        mutex_unlock(&st->gpio_lock);
        return ret;
    }
    mdelay(1);

    mutex_unlock(&st->gpio_lock);

    max11300_gpio_set(chip, offset, value);

    return ret;
}

```

Create a new `linux_5.4_max11300_driver` folder inside the `linux_5.4_rpi3_drivers` folder:

```
~/linux_5.4_rpi3_drivers$ mkdir linux_5.4_max11300_driver
```

Create the `max11300.c`, `max11300-base.c` and `max11300-base.h` files and also a Makefile file in the `linux_5.4_max11300_driver` folder, and add `max11300.o` and `max11300-base.o` to your Makefile `obj-m` variable, then build and deploy the modules to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_max11300_driver$ make
~/linux_5.4_rpi3_drivers/linux_5.4_max11300_driver$ make deploy
```

Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-hf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Listing 11-1: max11300-base.h

```
#ifndef __DRIVERS_IIO_DAC_max11300_BASE_H__
#define __DRIVERS_IIO_DAC_max11300_BASE_H__

#include <linux/types.h>
#include <linux/cache.h>
#include <linux/mutex.h>
#include <linux/gpio/driver.h>

struct max11300_state;

/* Masks for the Device Control (DCR) Register */
#define DCR_ADCCTL_CONTINUOUS_SWEEP (BIT(0) | BIT(1))
#define DCR_DACREF BIT(6)
#define BRST BIT(14)
#define RESET BIT(15)

/* Define register addresses */
#define DCR_ADDRESS 0x10
#define PORT_CFG_BASE_ADDRESS 0x20
#define PORT_ADC_DATA_BASE_ADDRESS 0x40
#define PORT_DAC_DATA_BASE_ADDRESS 0x60
#define DACPRSTDAT1_ADDRESS 0x16
#define GPO_DATA_15_TO_0_ADDRESS 0x0D
#define GPO_DATA_19_TO_16_ADDRESS 0x0E
#define GPI_DATA_15_TO_0_ADDRESS 0x0B
#define GPI_DATA_19_TO_16_ADDRESS 0x0C

/*
 * Declare a structure with pointers to the functions that will read and write
 * via SPI the registers of the MAX11300 device
 */
```

```

*/
struct max11300_rw_ops {
    int (*reg_write)(struct max11300_state *st, u8 reg, u16 value);
    int (*reg_read)(struct max11300_state *st, u8 reg, u16 *value);
    int (*reg_read_differential)(struct max11300_state *st, u8 reg, int *value);
};

/* Declare the global structure that will store the info of the device */
struct max11300_state {
    struct device *dev;
    const struct max11300_rw_ops *ops;
    struct gpio_chip gpiochip;
    struct mutex gpio_lock;
    u8 num_ports;
    u8 num_gpios;
    u8 gpio_offset[20];
    u8 gpio_offset_mode[20];
    u8 port_modes[20];
    u8 adc_range[20];
    u8 dac_range[20];
    u8 adc_reference[20];
    u8 adc_samples[20];
    u8 adc_negative_port[20];
    u8 tx_cmd;
    __be16 tx_msg;
    __be16 rx_msg;
};

int max11300_probe(struct device *dev, const char *name,
                  const struct max11300_rw_ops *ops);
int max11300_remove(struct device *dev);

#endif /* __DRIVERS_IIO_DAC_max11300_BASE_H__ */

```

Listing 11-2: maxim,max11300.h

```

#ifndef _DT_BINDINGS_MAXIM_MAX11300_H
#define _DT_BINDINGS_MAXIM_MAX11300_H

#define PORT_MODE_0      0
#define PORT_MODE_1      1
#define PORT_MODE_2      2
#define PORT_MODE_3      3
#define PORT_MODE_4      4
#define PORT_MODE_5      5
#define PORT_MODE_6      6
#define PORT_MODE_7      7
#define PORT_MODE_8      8
#define PORT_MODE_9      9
#define PORT_MODE_10     10
#define PORT_MODE_11     11
#define PORT_MODE_12     12

#define ADC_SAMPLES_1     0
#define ADC_SAMPLES_2     1
#define ADC_SAMPLES_4     2
#define ADC_SAMPLES_8     3
#define ADC_SAMPLES_16    4
#define ADC_SAMPLES_32    5
#define ADC_SAMPLES_64    6
#define ADC_SAMPLES_128   7

/* ADC voltage ranges */
#define ADC_VOLTAGE_RANGE_NOT_SELECTED 0
#define ADC_VOLTAGE_RANGE_PLUS10      1 // 0 to +5V range
#define ADC_VOLTAGE_RANGE_PLUSMINUS5  2 // -5V to +5V range
#define ADC_VOLTAGE_RANGE_MINUS10     3 // -10V to 0 range
#define ADC_VOLTAGE_RANGE_PLUS25      4 // 0 to +2.5 range

/* DAC voltage ranges mode 5*/
#define DAC_VOLTAGE_RANGE_NOT_SELECTED 0
#define DAC_VOLTAGE_RANGE_PLUS10      1
#define DAC_VOLTAGE_RANGE_PLUSMINUS5  2
#define DAC_VOLTAGE_RANGE_MINUS10     3

#endif /* _DT_BINDINGS_MAXIM_MAX11300_H */

```

Listing 11-3: max11300.c

```

#include "max11300-base.h"
#include <linux/bitops.h>
#include <linux/module.h>
#include <linux/of.h>
#include <linux/spi/spi.h>

/* Function to write MAX11300 registers */
static int max11300_reg_write(struct max11300_state *st, u8 reg, u16 val)
{
    struct spi_device *spi = container_of(st->dev, struct spi_device, dev);

    struct spi_transfer t[] = {
        {
            .tx_buf = &st->tx_cmd,
            .len = 1,
        }, {
            .tx_buf = &st->tx_msg,
            .len = 2,
        },
    };

    /* To transmit via SPI, the LSB bit of the command byte must be 0 */
    st->tx_cmd = (reg << 1);

    /*
     * In little endian CPUs the byte stored in the higher address of
     * the "val" variable (MSB of the DAC) will be stored in the lower address
     * of the "st->tx_msg" variable using cpu_to_be16()
     */
    st->tx_msg = cpu_to_be16(val);

    return spi_sync_transfer(spi, t, ARRAY_SIZE(t));
}

/* Function to read MAX11300 registers in SE mode */
static int max11300_reg_read(struct max11300_state *st, u8 reg, u16 *value)
{
    struct spi_device *spi = container_of(st->dev, struct spi_device, dev);
    int ret;

    struct spi_transfer t[] = {
        {
            .tx_buf = &st->tx_cmd,
            .len = 1,
        }, {
            .rx_buf = &st->rx_msg,
            .len = 2,
        },
    };

    dev_info(st->dev, "read SE channel\n");

```



```

/* to receive via SPI the LSB bit of the command byte must be 1 */
st->tx_cmd = ((reg << 1) | 1);

ret = spi_sync_transfer(spi, t, ARRAY_SIZE(t));
if (ret < 0)
    return ret;

/*
 * In little endian CPUs the first byte (MSB of the ADC) received via
 * SPI (in BE format) is stored in the lower address of "st->rx_msg"
 * variable. This byte is copied to the higher address of the "value"
 * variable using be16_to_cpu(). The second byte received via SPI is
 * copied from the higher address of "st->rx_msg" to the lower address
 * of the "value" variable in little endian CPUs.
 * In big endian CPUs the addresses are not swapped.
 */
*value = be16_to_cpu(st->rx_msg);

return 0;
}

/* Function to read MAX11300 registers in differential mode (2's complement) */
static int max11300_reg_read_differential(struct max11300_state *st, u8 reg, int *value)
{
    struct spi_device *spi = container_of(st->dev, struct spi_device, dev);
    int ret;

    struct spi_transfer t[] = {
        {
            .tx_buf = &st->tx_cmd,
            .len = 1,
        }, {
            .rx_buf = &st->rx_msg,
            .len = 2,
        },
    };

    dev_info(st->dev, "read differential channel\n");

    /* LSB of command byte has to be 1 */
    st->tx_cmd = ((reg << 1) | 1);

    ret = spi_sync_transfer(spi, t, ARRAY_SIZE(t));
    if (ret < 0)
        return ret;

    /*
     * Extend to an int 2's complement value the received SPI value in 2's
     * complement value which is stored in the "st->rx_msg" variable
     */
    *value = sign_extend32(be16_to_cpu(st->rx_msg), 11);

    return 0;
}

```

```

/*
 * Initialize struct max11300_rw_ops with callback functions
 * that will write/read via SPI the MAX11300 registers
 */
static const struct max11300_rw_ops max11300_rw_ops = {
    .reg_write = max11300_reg_write,
    .reg_read = max11300_reg_read,
    .reg_read_differential = max11300_reg_read_differential,
};

static int max11300_spi_probe(struct spi_device *spi)
{
    const struct spi_device_id *id = spi_get_device_id(spi);

    return max11300_probe(&spi->dev, id->name, &max11300_rw_ops);
}

static int max11300_spi_remove(struct spi_device *spi)
{
    return max11300_remove(&spi->dev);
}

static const struct spi_device_id max11300_spi_ids[] = {
    { .name = "max11300", },
    {}
};
MODULE_DEVICE_TABLE(spi, max11300_spi_ids);

static const struct of_device_id max11300_of_match[] = {
    { .compatible = "maxim,max11300", },
    {},
};
MODULE_DEVICE_TABLE(of, max11300_of_match);

static struct spi_driver max11300_spi_driver = {
    .driver = {
        .name = "max11300",
        .of_match_table = of_match_ptr(max11300_of_match),
    },
    .probe = max11300_spi_probe,
    .remove = max11300_spi_remove,
    .id_table = max11300_spi_ids,
};
module_spi_driver(max11300_spi_driver);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("Maxim max11300 multi-port converters");
MODULE_LICENSE("GPL v2");

```

Listing 11-4: max11300-base.c

```
#include <linux/bitops.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>
#include <linux/module.h>
#include <linux/mutex.h>
#include <linux/of.h>
#include <linux/property.h>

#include <dt-bindings/iio/maxim,max11300.h>

#include "max11300-base.h"

/*
 * struct gpio_chip get callback function.
 * It gets the input value of the GPIO line (0=low, 1=high)
 * accessing to the GPI_DATA registers of max11300
 */
static int max11300_gpio_get(struct gpio_chip *chip, unsigned int offset)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    int ret = 0;
    u16 read_val;
    u8 reg;
    int val;

    mutex_lock(&st->gpio_lock);

    dev_info(st->dev, "The GPIO input is get\n");

    if (st->gpio_offset_mode[offset] == PORT_MODE_3)
        dev_info(st->dev, "the gpio %d cannot be configured in input mode\n", offset);

    /* for GPIOs from 16 to 19 ports */
    if (st->gpio_offset[offset] > 0x0F) {
        reg = GPI_DATA_19_TO_16_ADDRESS;
        ret = st->ops->reg_read(st, reg, &read_val);
        if (ret)
            goto err_unlock;

        val = (int) (read_val);
        val = val << 16;

        if (val & BIT(st->gpio_offset[offset]))
            val = 1;
        else
            val = 0;

        mutex_unlock(&st->gpio_lock);
        return val;
    }
    else {
```

```

        reg = GPI_DATA_15_TO_0_ADDRESS;
        ret = st->ops->reg_read(st, reg, &read_val);
        if (ret)
            goto err_unlock;

        val = (int) read_val;

        if(val & BIT(st->gpio_offset[offset]))
            val = 1;
        else
            val = 0;

        mutex_unlock(&st->gpio_lock);
        return val;
    }

err_unlock:
    mutex_unlock(&st->gpio_lock);
    return ret;
}

/*
 * struct gpio_chip set callback function.
 * It sets the output value of the GPIO line in
 * GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO_DATA registers of max11300
 */
static void max11300_gpio_set(struct gpio_chip *chip, unsigned int offset, int value)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    u8 reg;
    unsigned int val = 0;

    mutex_lock(&st->gpio_lock);

    dev_info(st->dev, "The GPIO ouput is set\n");

    if (st->gpio_offset_mode[offset] == PORT_MODE_1)
        dev_info(st->dev, "the gpio %d cannot accept this output\n", offset);

    if (value == 1 && (st->gpio_offset[offset] > 0x0F)) {
        dev_info(st->dev,
            "The GPIO ouput is set high and port_number is %d. Pin is > 0x0F\n",
            st->gpio_offset[offset]);
        val |= BIT(st->gpio_offset[offset]);
        val = val >> 16;
        reg = GPO_DATA_19_TO_16_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 0 && (st->gpio_offset[offset] > 0x0F)) {
        dev_info(st->dev,
            "The GPIO ouput is set low and port_number is %d. Pin is > 0x0F\n",
            st->gpio_offset[offset]);
        val &= ~BIT(st->gpio_offset[offset]);
    }
}

```

```

        val = val >> 16;
        reg = GPO_DATA_19_TO_16_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 1 && (st->gpio_offset[offset] < 0x0F)) {
        dev_info(st->dev,
            "The GPIO ouput is set high and port_number is %d. Pin is < 0x0F\n",
            st->gpio_offset[offset]);
        val |= BIT(st->gpio_offset[offset]);
        reg = GPO_DATA_15_TO_0_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else if (value == 0 && (st->gpio_offset[offset] < 0x0F)) {
        dev_info(st->dev,
            "The GPIO ouput is set low and port_number is %d. Pin is < 0x0F\n",
            st->gpio_offset[offset]);
        val &= ~BIT(st->gpio_offset[offset]);
        reg = GPO_DATA_15_TO_0_ADDRESS;
        st->ops->reg_write(st, reg, val);
    }
    else
        dev_info(st->dev, "the gpio %d cannot accept this value\n", offset);

    mutex_unlock(&st->gpio_lock);
}

/*
 * struct gpio_chip direction_input callback function.
 * It configures the GPIO port as an input (GPI)
 * writing to the PORT_CFG register of max11300
 */
static int max11300_gpio_direction_input(struct gpio_chip *chip, unsigned int offset)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    int ret;
    u8 reg;
    u16 port_mode, val;

    mutex_lock(&st->gpio_lock);

    dev_info(st->dev, "The GPIO is set as an input\n");

    /* Get the port number stored in the GPIO offset */
    if (st->gpio_offset_mode[offset] == PORT_MODE_3)
        dev_info(st->dev,
            "Error.The gpio %d only can be set in output mode\n",
            offset);

    /* Set the logic 1 input above 2.5V level*/
    val = 0x0fff;

    /* Store the GPIO threshold value in the port DAC register */
    reg = PORT_DAC_DATA_BASE_ADDRESS + st->gpio_offset[offset];
    ret = st->ops->reg_write(st, reg, val);
}

```

```

    if (ret)
        goto err_unlock;

    /* Configure the port as GPI */
    reg = PORT_CFG_BASE_ADDRESS + st->gpio_offset[offset];
    port_mode = (1 << 12);
    ret = st->ops->reg_write(st, reg, port_mode);
    if (ret)
        goto err_unlock;

    mdelay(1);

err_unlock:
    mutex_unlock(&st->gpio_lock);

    return ret;
}

/*
 * struct gpio_chip direction_output callback function.
 * It configures the GPIO port as an output (GPO) writing to
 * the PORT_CFG register of max11300 and sets output value of the
 * GPIO line in GPIO ACTIVE_HIGH mode (0=low, 1=high)
 * writing to the GPO data registers of max11300
 */
static int max11300_gpio_direction_output(struct gpio_chip *chip,
                                         unsigned int offset, int value)
{
    struct max11300_state *st = gpiochip_get_data(chip);
    int ret;
    u8 reg;
    u16 port_mode, val;

    mutex_lock(&st->gpio_lock);

    dev_info(st->dev, "The GPIO is set as an output\n");

    if (st->gpio_offset_mode[offset] == PORT_MODE_1)
        dev_info(st->dev,
                 "the gpio %d only can be set in input mode\n",
                 offset);

    /* GPIO output high is 3.3V */
    val = 0x0547;

    reg = PORT_DAC_DATA_BASE_ADDRESS + st->gpio_offset[offset];
    ret = st->ops->reg_write(st, reg, val);
    if (ret) {
        mutex_unlock(&st->gpio_lock);
        return ret;
    }
    mdelay(1);
    reg = PORT_CFG_BASE_ADDRESS + st->gpio_offset[offset];
    port_mode = (3 << 12);

```

```

    ret = st->ops->reg_write(st, reg, port_mode);
    if (ret) {
        mutex_unlock(&st->gpio_lock);
        return ret;
    }
    mdelay(1);

    mutex_unlock(&st->gpio_lock);

    max11300_gpio_set(chip, offset, value);

    return ret;
}

/*
 * Initialize the MAX11300 gpio controller (struct gpio_chip)
 * and register it to the kernel
 */
static int max11300_gpio_init(struct max11300_state *st)
{
    if (!st->num_gpios)
        return 0;

    st->gpiochip.label = "gpio-max11300";
    st->gpiochip.base = -1;
    st->gpiochip.ngpio = st->num_gpios;
    st->gpiochip.parent = st->dev;
    st->gpiochip.can_sleep = true;
    st->gpiochip.direction_input = max11300_gpio_direction_input;
    st->gpiochip.direction_output = max11300_gpio_direction_output;
    st->gpiochip.get = max11300_gpio_get;
    st->gpiochip.set = max11300_gpio_set;
    st->gpiochip.owner = THIS_MODULE;

    mutex_init(&st->gpio_lock);

    /* register a gpio_chip */
    return gpiochip_add_data(&st->gpiochip, st);
}

/*
 * Configure the port configuration registers of each port with the values
 * retrieved from the DT properties. These DT values were read and stored in
 * the device global structure using the max11300_alloc_ports() function.
 * The ports in GPIO mode will be configured in the gpiochip.direction_input
 * and gpiochip.direction_output callback functions.
 */
static int max11300_set_port_modes(struct max11300_state *st)
{
    const struct max11300_rw_ops *ops = st->ops;
    int ret;
    unsigned int i;
    u8 reg;
    u16 adc_range, dac_range, adc_reference, adc_samples, adc_negative_port;

```

```

u16 val, port_mode;
struct iio_dev *iio_dev = iio_priv_to_dev(st);

mutex_lock(&iio_dev->mlock);

for (i = 0; i < st->num_ports; i++) {
    switch (st->port_modes[i]) {
        case PORT_MODE_5: case PORT_MODE_6:
            reg = PORT_CFG_BASE_ADDRESS + i;
            adc_reference = st->adc_reference[i];
            port_mode = (st->port_modes[i] << 12);
            dac_range = (st->dac_range[i] << 8);

            dev_info(st->dev,
                "the value of adc cfg addr for channel %d in port mode %d is %x\n",
                    i, st->port_modes[i], reg);

            if ((st->port_modes[i]) == PORT_MODE_5)
                val = (port_mode | dac_range);
            else
                val = (port_mode | dac_range | adc_reference);

            dev_info(st->dev, "the channel %d is set in port mode %d\n",
                i, st->port_modes[i]);
            dev_info(st->dev,
                "the value of adc cfg val for channel %d in port mode %d is %x\n",
                    i, st->port_modes[i], val);

            ret = ops->reg_write(st, reg, val);
            if (ret)
                goto err_unlock;

            mdelay(1);
            break;
        case PORT_MODE_7:
            reg = PORT_CFG_BASE_ADDRESS + i;
            port_mode = (st->port_modes[i] << 12);
            adc_range = (st->adc_range[i] << 8);
            adc_reference = st->adc_reference[i];
            adc_samples = (st->adc_samples[i] << 5);

            dev_info(st->dev,
                "the value of adc cfg addr for channel %d in port mode %d is %x\n",
                    i, st->port_modes[i], reg);

            val = (port_mode | adc_range | adc_reference | adc_samples);

            dev_info(st->dev,
                "the channel %d is set in port mode %d\n",
                    i, st->port_modes[i]);
            dev_info(st->dev,
                "the value of adc cfg val for channel %d in port mode %d is %x\n",
                    i, st->port_modes[i], val);
    }
}

```



```

        ret = ops->reg_write(st, reg, val);
        if (ret)
            goto err_unlock;

        mdelay(1);

        break;
case PORT_MODE_8:
    reg = PORT_CFG_BASE_ADDRESS + i;
    port_mode = (st->port_modes[i] << 12);
    adc_range = (st->adc_range[i] << 8);
    adc_reference = st->adc_reference[i];
    adc_samples = (st->adc_samples[i] << 5);
    adc_negative_port = st->adc_negative_port[i];

    dev_info(st->dev,
        "the value of adc cfg addr for channel %d in port mode %d is %x\n",
            i, st->port_modes[i], reg);

    val = (port_mode | adc_range | adc_reference | adc_samples | adc_negative_
port);

    dev_info(st->dev,
        "the channel %d is set in port mode %d\n",
            i, st->port_modes[i]);
    dev_info(st->dev,
        "the value of adc cfg val for channel %d in port mode %d is %x\n",
            i, st->port_modes[i], val);

    ret = ops->reg_write(st, reg, val);
    if (ret)
        goto err_unlock;

    mdelay(1);
    break;
case PORT_MODE_9: case PORT_MODE_10:
    reg = PORT_CFG_BASE_ADDRESS + i;
    port_mode = (st->port_modes[i] << 12);
    adc_range = (st->adc_range[i] << 8);
    adc_reference = st->adc_reference[i];

    dev_info(st->dev,
        "the value of adc cfg addr for channel %d in port mode %d is %x\n",
            i, st->port_modes[i], reg);

    val = (port_mode | adc_range | adc_reference);

    dev_info(st->dev,
        "the channel %d is set in port mode %d\n",
            i, st->port_modes[i]);
    dev_info(st->dev,
        "the value of adc cfg val for channel %d in port mode %d is %x\n",
            i, st->port_modes[i], val);

```

```

        ret = ops->reg_write(st, reg, val);
        if (ret)
            goto err_unlock;

        mdelay(1);
        break;
case PORT_MODE_0:
    dev_info(st->dev, "the port %d is set in default port mode_0\n", i);
    break;
case PORT_MODE_1:
    dev_info(st->dev, "the port %d is set in port mode_1\n", i);
    break;
case PORT_MODE_3:
    dev_info(st->dev, "the port %d is set in port mode_3\n", i);
    break;
default:
    dev_info(st->dev, "bad port mode is selected\n");
    return -EINVAL;
    }
}

err_unlock:
    mutex_unlock(&iio_dev->mlock);
    return ret;
}

/* IIO writing callback function */
static int max11300_write_dac(struct iio_dev *iio_dev, struct iio_chan_spec const *chan,
                             int val, int val2, long mask)
{
    struct max11300_state *st = iio_priv(iio_dev);
    u8 reg;
    int ret;

    reg = (PORT_DAC_DATA_BASE_ADDRESS + chan->channel);

    dev_info(st->dev, "the DAC data register is %x\n", reg);
    dev_info(st->dev, "the value in the DAC data register is %x\n", val);

    switch (mask) {
case IIO_CHAN_INFO_RAW:
        if (!chan->output)
            return -EINVAL;

        mutex_lock(&iio_dev->mlock);
        ret = st->ops->reg_write(st, reg, val);
        mutex_unlock(&iio_dev->mlock);
        break;
default:
        return -EINVAL;
    }

    return ret;
}

```

```

/* IIO reading callback function */
static int max11300_read_adc(struct iio_dev *iio_dev,
                             struct iio_chan_spec const *chan,
                             int *val, int *val2, long m)
{
    struct max11300_state *st = iio_priv(iio_dev);
    u16 read_val_se;
    int read_val_dif;
    u8 reg;
    int ret;

    reg = PORT_ADC_DATA_BASE_ADDRESS + chan->channel;

    switch (m) {
    case IIO_CHAN_INFO_RAW:
        mutex_lock(&iio_dev->mlock);

        if (!chan->output && ((chan->address == PORT_MODE_7) || (chan->address == PORT_
MODE_6))) {
            ret = st->ops->reg_read(st, reg, &read_val_se);
            if (ret)
                goto unlock;
            *val = (int) read_val_se;
        }
        else if (!chan->output && (chan->address == PORT_MODE_8)) {
            ret = st->ops->reg_read_differential(st, reg, &read_val_dif);
            if (ret)
                goto unlock;
            *val = read_val_dif;
        }
        else {
            ret = -EINVAL;
            goto unlock;
        }

        ret = IIO_VAL_INT;
        break;
    default:
        ret = -EINVAL;
    }

unlock:
    mutex_unlock(&iio_dev->mlock);
    return ret;
}

/* Create kernel hooks to read/write IIO sysfs attributes from user space */
static const struct iio_info max11300_info = {
    .read_raw = max11300_read_adc,
    .write_raw = max11300_write_dac,
};

/* DAC with positive voltage range */

```

```

static void max11300_setup_port_5_mode(struct iio_dev *iio_dev,
                                     struct iio_chan_spec *chan, bool output,
                                     unsigned int id, unsigned long port_mode)
{
    chan->type = IIO_VOLTAGE;
    chan->indexed = 1;
    chan->address = port_mode;
    chan->output = output;
    chan->channel = id;
    chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
    chan->scan_type.sign = 'u';
    chan->scan_type.realbits = 12;
    chan->scan_type.storagebits = 16;
    chan->scan_type.endianness = IIO_BE;
    chan->extend_name = "mode_5_DAC";
}

/* DAC with positive voltage range */
static void max11300_setup_port_6_mode(struct iio_dev *iio_dev,
                                     struct iio_chan_spec *chan, bool output,
                                     unsigned int id, unsigned long port_mode)
{
    chan->type = IIO_VOLTAGE;
    chan->indexed = 1;
    chan->address = port_mode;
    chan->output = output;
    chan->channel = id;
    chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
    chan->scan_type.sign = 'u';
    chan->scan_type.realbits = 12;
    chan->scan_type.storagebits = 16;
    chan->scan_type.endianness = IIO_BE;
    chan->extend_name = "mode_6_DAC_ADC";
}

/* ADC in SE mode with positive voltage range and straight binary */
static void max11300_setup_port_7_mode(struct iio_dev *iio_dev,
                                     struct iio_chan_spec *chan, bool output,
                                     unsigned int id, unsigned long port_mode)
{
    chan->type = IIO_VOLTAGE;
    chan->indexed = 1;
    chan->address = port_mode;
    chan->output = output;
    chan->channel = id;
    chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
    chan->scan_type.sign = 'u';
    chan->scan_type.realbits = 12;
    chan->scan_type.storagebits = 16;
    chan->scan_type.endianness = IIO_BE;
    chan->extend_name = "mode_7_ADC";
}

/* ADC in differential mode with 2's complement value */

```

```

static void max11300_setup_port_8_mode(struct iio_dev *iio_dev,
                                     struct iio_chan_spec *chan, bool output,
                                     unsigned id, unsigned id2,
                                     unsigned int port_mode)
{
    chan->type = IIO_VOLTAGE;
    chan->differential = 1,
    chan->address = port_mode;
    chan->indexed = 1;
    chan->output = output;
    chan->channel = id;
    chan->channel2 = id2;
    chan->info_mask_separate = BIT(IIO_CHAN_INFO_RAW);
    chan->scan_type.sign = 's';
    chan->scan_type.realbits = 12;
    chan->scan_type.storagebits = 16;
    chan->scan_type.endianness = IIO_BE;
    chan->extend_name = "mode_8_ADC";
}

/*
 * This function will allocate and configure the iio channels of the iio device.
 * It will also read the DT properties of each port (channel) and will store them
 * in the device global structure
 */
static int max11300_alloc_ports(struct max11300_state *st)
{
    unsigned int i, curr_port = 0, num_ports = st->num_ports, port_mode_6_count = 0, offset =
0;
    st->num_gpios = 0;

    /* Recover the iio device from the global structure */
    struct iio_dev *iio_dev = iio_priv_to_dev(st);

    /* Pointer to the storage of the specs of all the iio channels */
    struct iio_chan_spec *ports;

    /* Pointer to struct fwnode_handle that allows a device description object */
    struct fwnode_handle *child;

    u32 reg, tmp;
    int ret;

    /*
     * Walks for each MAX11300 child node in the DT. If there is an error, then
     * walks to the following node (continue)
     */
    device_for_each_child_node(st->dev, child) {
        ret = fwnode_property_read_u32(child, "reg", &reg);
        if (ret || reg >= ARRAY_SIZE(st->port_modes))
            continue;

        /*
         * Store the value of the DT "port,mode" property in the global struct

```

```

    * to know the mode of each port in other functions of the driver
    */
    ret = fwnode_property_read_u32(child, "port-mode", &tmp);
    if (!ret)
        st->port_modes[reg] = tmp;

    /* all the DT nodes should include the port-mode property */
    else {
        dev_info(st->dev, "port mode is not found\n");
        continue;
    }

    /*
     * You will store other DT properties depending
     * of the used "port,mode" property
     */
    switch (st->port_modes[reg]) {
    case PORT_MODE_7:
        ret = fwnode_property_read_u32(child, "adc-range", &tmp);
        if (!ret)
            st->adc_range[reg] = tmp;
        else
            dev_info(st->dev, "Get default ADC range\n");

        ret = fwnode_property_read_u32(child, "AVR", &tmp);
        if (!ret)
            st->adc_reference[reg] = tmp;
        else
            dev_info(st->dev, "Get default internal ADC reference\n");

        ret = fwnode_property_read_u32(child, "adc-samples", &tmp);
        if (!ret)
            st->adc_samples[reg] = tmp;
        else
            dev_info(st->dev, "Get default internal ADC sampling\n");

        dev_info(st->dev, "the channel %d is set in port mode %d\n",
            reg, st->port_modes[reg]);
        break;
    case PORT_MODE_8:
        ret = fwnode_property_read_u32(child, "adc-range", &tmp);
        if (!ret)
            st->adc_range[reg] = tmp;
        else
            dev_info(st->dev, "Get default ADC range\n");

        ret = fwnode_property_read_u32(child, "AVR", &tmp);
        if (!ret)
            st->adc_reference[reg] = tmp;
        else
            dev_info(st->dev, "Get default internal ADC reference\n");

        ret = fwnode_property_read_u32(child, "adc-samples", &tmp);
        if (!ret)

```

```

        st->adc_samples[reg] = tmp;
    else
        dev_info(st->dev, "Get default internal ADC sampling\n");

    ret = fwnode_property_read_u32(child, "negative-input", &tmp);
    if (!ret)
        st->adc_negative_port[reg] = tmp;
    else {
        dev_info(st->dev, "Bad value for negative ADC channel\n");
        return -EINVAL;
    }
    dev_info(st->dev, "the channel %d is set in port mode %d\n",
        reg, st->port_modes[reg]);
    break;
case PORT_MODE_9: case PORT_MODE_10:
    ret = fwnode_property_read_u32(child, "adc-range", &tmp);
    if (!ret)
        st->adc_range[reg] = tmp;
    else
        dev_info(st->dev, "Get default ADC range\n");

    ret = fwnode_property_read_u32(child, "AVR", &tmp);
    if (!ret)
        st->adc_reference[reg] = tmp;
    else
        dev_info(st->dev, "Get default internal ADC reference\n");
    dev_info(st->dev, "the channel %d is set in port mode %d\n",
        reg, st->port_modes[reg]);
    break;
case PORT_MODE_5: case PORT_MODE_6:
    ret = fwnode_property_read_u32(child, "dac-range", &tmp);
    if (!ret)
        st->dac_range[reg] = tmp;
    else
        dev_info(st->dev, "Get default DAC range\n");

    /*
     * A port in mode 6 will generate two IIO sysfs entries,
     * one for writing the DAC port, and another for reading
     * the ADC port
     */
    if ((st->port_modes[reg]) == PORT_MODE_6) {
        ret = fwnode_property_read_u32(child, "AVR", &tmp);
        if (!ret)
            st->adc_reference[reg] = tmp;
        else
            dev_info(st->dev, "Get default internal ADC reference\n");

        /*
         * Get the number of ports set in mode_6 to allocate
         * space for the related iio channels
         */
        port_mode_6_count++;
        dev_info(st->dev, "there are %d channels in mode_6\n",

```

```

        port_mode_6_count);
    }

    dev_info(st->dev, "the channel %d is set in port mode %d\n",
             reg, st->port_modes[reg]);
    break;
/* The port is configured as a GPI in the DT */
case PORT_MODE_1:
    dev_info(st->dev, "the channel %d is set in port mode %d\n",
             reg, st->port_modes[reg]);

    /*
     * link the gpio offset with the port number,
     * starting with offset = 0
     */
    st->gpio_offset[offset] = reg;

    /*
     * store the port_mode for each gpio offset,
     * starting with offset = 0
     */
    st->gpio_offset_mode[offset] = PORT_MODE_1;

    dev_info(st->dev,
             "the gpio number %d is using the gpio offset number %d\n",
             st->gpio_offset[offset], offset);

    /*
     * Increment the gpio offset and number
     * of configured ports as GPIOs
     */
    offset++;
    st->num_gpios++;
    break;
/* The port is configured as a GPO in the DT */
case PORT_MODE_3:
    dev_info(st->dev, "the channel %d is set in port mode %d\n",
             reg, st->port_modes[reg]);

    /*
     * link the gpio offset with the port number,
     * starting with offset = 0
     */
    st->gpio_offset[offset] = reg;

    /*
     * Store the port_mode for each gpio offset,
     * starting with offset = 0
     */
    st->gpio_offset_mode[offset] = PORT_MODE_3;

    dev_info(st->dev,
             "the gpio number %d is using the gpio offset number %d\n",
             st->gpio_offset[offset], offset);

```



```

        /*
        * Increment the gpio offset and
        * number of configured ports as GPIOs
        */
        offset++;
        st->num_gpios++;
        break;
    case PORT_MODE_0:
        dev_info(st->dev, "the channel %d is set in default port mode_0\n", reg);
        break;
    default:
        dev_info(st->dev, "bad port mode for channel %d\n", reg);
    }
}

/*
 * Allocate space for the storage of all the IIO channels specs.
 * Returns a pointer to this storage
 */
ports = devm_kcalloc(st->dev, num_ports + port_mode_6_count, sizeof(*ports), GFP_KERNEL);
if (!ports)
    return -ENOMEM;

/*
 * i is the number of the channel, &ports[curr_port] is a pointer variable that
 * will store the "iio_chan_spec structure" address of each port
 */
for (i = 0; i < num_ports; i++) {
    switch (st->port_modes[i]) {
        case PORT_MODE_5:
            dev_info(st->dev, "the port %d is configured as MODE 5\n", i);
            max11300_setup_port_5_mode(iio_dev, &ports[curr_port],
                                       true, i, PORT_MODE_5); // true = out
            curr_port++;
            break;
        case PORT_MODE_6:
            dev_info(st->dev, "the port %d is configured as MODE 6\n", i);
            max11300_setup_port_6_mode(iio_dev, &ports[curr_port],
                                       true, i, PORT_MODE_6); // true = out
            curr_port++;
            max11300_setup_port_6_mode(iio_dev, &ports[curr_port],
                                       false, i, PORT_MODE_6); // false = in
            curr_port++;
            break;
        case PORT_MODE_7:
            dev_info(st->dev, "the port %d is configured as MODE 7\n", i);
            max11300_setup_port_7_mode(iio_dev, &ports[curr_port],
                                       false, i, PORT_MODE_7); // false = in
            curr_port++;
            break;
        case PORT_MODE_8:

```

```

        dev_info(st->dev, "the port %d is configured as MODE 8\n", i);
        max11300_setup_port_8_mode(iio_dev, &ports[curr_port],
                                   false, i, st->adc_negative_port[i],
                                   PORT_MODE_8); // false = in
        curr_port++;
        break;
    case PORT_MODE_0:
        dev_info(st->dev, "the channel is set in default port mode_0\n");
        break;
    case PORT_MODE_1:
        dev_info(st->dev, "the channel %d is set in port mode_1\n", i);
        break;
    case PORT_MODE_3:
        dev_info(st->dev, "the channel %d is set in port mode_3\n", i);
        break;
    default:
        dev_info(st->dev, "bad port mode for channel %d\n", i);
    }
}

iio_dev->num_channels = curr_port;
iio_dev->channels = ports;

return 0;
}

int max11300_probe(struct device *dev, const char *name, const struct max11300_rw_ops *ops)
{
    /* Create an iio device */
    struct iio_dev *iio_dev;

    /* Create the global structure that will store the info of the device */
    struct max11300_state *st;

    u16 write_val;
    u16 read_val;
    u8 reg;
    int ret;

    write_val = 0;

    dev_info(dev, "max11300_probe() function is called\n");

    /* Allocates memory for the IIO device */
    iio_dev = devm_iio_device_alloc(dev, sizeof(*st));
    if (!iio_dev)
        return -ENOMEM;

    /* Link the global data structure with the iio device */
    st = iio_priv(iio_dev);

    /* Store in the global structure the spi device */
    st->dev = dev;

```

```

/*
 * Store in the global structure the pointer to the
 * MAX11300 SPI read and write functions
 */
st->ops = ops;

/* Set the number of ports of the MAX11300 device */
st->num_ports = 20;

/* Link the spi device with the iio device */
dev_set_drvdata(dev, iio_dev);

iio_dev->dev.parent = dev;
iio_dev->name = name;

/*
 * Store the address of the iio_info structure,
 * which contains pointer variables
 * to the IIO write/read callbacks
 */
iio_dev->info = &max11300_info;
iio_dev->modes = INDIO_DIRECT_MODE;

/* Reset the MAX11300 device */
reg = DCR_ADDRESS;
dev_info(st->dev, "the value of DCR_ADDRESS is %x\n", reg);
write_val = RESET;
dev_info(st->dev, "the value of reset is %x\n", write_val);
ret = ops->reg_write(st, reg, write_val);
if (ret != 0)
    goto error;

/* Return MAX11300 Device ID */
reg = 0x00;
ret = ops->reg_read(st, reg, &read_val);
if (ret != 0)
    goto error;
dev_info(st->dev, "the value of device ID is %x\n", read_val);

/* Configure DACREF and ADCCTL */
reg = DCR_ADDRESS;
write_val = (DCR_ADCCTL_CONTINUOUS_SWEEP | DCR_DACREF);
dev_info(st->dev, "the value of DACREF_CONT_SWEEP is %x\n", write_val);
ret = ops->reg_write(st, reg, write_val);
udelay(200);
if (ret)
    goto error;
dev_info(dev, "the setup of the device is done\n");

/* Configure the IIO channels of the device */
ret = max11300_alloc_ports(st);
if (ret)
    goto error;

```

```
ret = max11300_set_port_modes(st);
if (ret)
    goto error_reset_device;

ret = iio_device_register(iio_dev);
if (ret)
    goto error;

ret = max11300_gpio_init(st);
if (ret)
    goto error_dev_unregister;

return 0;

error_dev_unregister:
    iio_device_unregister(iio_dev);

error_reset_device:
    /* reset the device */
    reg = DCR_ADDRESS;
    write_val = RESET;
    ret = ops->reg_write(st, reg, write_val);
    if (ret != 0)
        return ret;

error:
    return ret;
}
EXPORT_SYMBOL_GPL(max11300_probe);

int max11300_remove(struct device *dev)
{
    struct iio_dev *iio_dev = dev_get_drvdata(dev);

    iio_device_unregister(iio_dev);

    return 0;
}
EXPORT_SYMBOL_GPL(max11300_remove);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("Maxim max11300 multi-port converters");
MODULE_LICENSE("GPL v2");
```

LAB 11.1 driver demonstration

Libgpiod provides a C library and simple tools for interacting with the Linux GPIO character devices. The GPIO sysfs interface is deprecated from Linux 4.8 for these libgpiod tools. The C library encapsulates the `ioctl()` calls and data structures using a straightforward API. For more information see: <https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/about/>

Connect your Raspberry Pi to the Internet and download libgpiod library and tools:

```
root@raspberrypi:/home# sudo apt-get install gpiod libgpiod-dev libgpiod-doc
```

The tools provided with libgpiod allow the access to the GPIO driver from the command line. There are six commands included in the libgpiod tools:

- **gpiodetect**: Lists all the gpiochips present on the system, their names, labels and the number of GPIO lines. In the lab, the MAX11300 gpio chip will be shown with the name of gpiochip3.
- **gpioinfo**: Lists all the lines of the specified gpiochips, their names, consumers, directions, active states and additional flags.
- **gpioget**: Reads the values of the specified GPIO lines. This tool will call the `gpiochip.direction_input` and `gpiochip.get` callback functions which were declared in struct `gpio_chip`.
- **gpioset**: Sets the values of the specified GPIO lines. This tool will call the `gpiochip.direction_output` callback function which was declared in struct `gpio_chip`.
- **gpiofind**: Finds the gpiochip name and line offset given the line name.
- **gpiomon**: Waits for events on GPIO lines, specifies which events to watch, how many events to process before exiting or if the events should be reported to the console.

Follow the next instructions to test the driver:

Load the modules:

```
root@raspberrypi:/home/pi# insmod max11300-base.ko
```

```
root@raspberrypi:/home/pi# insmod max11300.ko
max11300 spi0.0: max11300_probe() function is called
max11300 spi0.0: the value of DCR_ADDRESS is 10
max11300 spi0.0: the value of reset is 8000
max11300 spi0.0: read SE channel
max11300 spi0.0: the value of device ID is 424
max11300 spi0.0: the value of DACREF_CONT_SWEEP is 43
max11300 spi0.0: the setup of the device is done
```

```
[...]
```

```

max11300 spi0.0: the channel 0 is set in port mode 7
max11300 spi0.0: the value of adc cfg val for channel 0 in port mode 7 is 7100
max11300 spi0.0: the value of adc cfg addr for channel 1 in port mode 7 is 21
max11300 spi0.0: the channel 1 is set in port mode 7
max11300 spi0.0: the value of adc cfg val for channel 1 in port mode 7 is 71e0
max11300 spi0.0: the value of adc cfg addr for channel 2 in port mode 5 is 22
max11300 spi0.0: the channel 2 is set in port mode 5
max11300 spi0.0: the value of adc cfg val for channel 2 in port mode 5 is 5100
max11300 spi0.0: the value of adc cfg addr for channel 3 in port mode 5 is 23
max11300 spi0.0: the channel 3 is set in port mode 5
max11300 spi0.0: the value of adc cfg val for channel 3 in port mode 5 is 5100
max11300 spi0.0: the value of adc cfg addr for channel 4 in port mode 8 is 24
max11300 spi0.0: the channel 4 is set in port mode 8
max11300 spi0.0: the value of adc cfg val for channel 4 in port mode 8 is 8105
max11300 spi0.0: the value of adc cfg addr for channel 5 in port mode 9 is 25
max11300 spi0.0: the channel 5 is set in port mode 9
max11300 spi0.0: the value of adc cfg val for channel 5 in port mode 9 is 9100
max11300 spi0.0: the value of adc cfg addr for channel 6 in port mode 6 is 26
max11300 spi0.0: the channel 6 is set in port mode 6
max11300 spi0.0: the value of adc cfg val for channel 6 in port mode 6 is 6100
max11300 spi0.0: the port 7 is set in port mode_1
max11300 spi0.0: the port 8 is set in port mode_3
max11300 spi0.0: the port 9 is set in default port mode_0
max11300 spi0.0: the port 10 is set in default port mode_0
max11300 spi0.0: the port 11 is set in default port mode_0
max11300 spi0.0: the port 12 is set in default port mode_0
max11300 spi0.0: the port 13 is set in default port mode_0
max11300 spi0.0: the port 14 is set in default port mode_0
max11300 spi0.0: the port 15 is set in default port mode_0
max11300 spi0.0: the port 16 is set in default port mode_0
max11300 spi0.0: the port 17 is set in default port mode_0
max11300 spi0.0: the port 18 is set in port mode_1
max11300 spi0.0: the port 19 is set in port mode_3
root@raspberrypi:/home#

```

```
root@raspberrypi:/home/pi# cd /sys/bus/iio/devices/iio:device0
```

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# ls
```

dev	name	power
in_voltage0_mode_7_ADC_raw	of_node	subsystem
in_voltage1_mode_7_ADC_raw	out_voltage2_mode_5_DAC_raw	uevent
in_voltage4-voltage5_mode_8_ADC_raw	out_voltage3_mode_5_DAC_raw	
in_voltage6_mode_6_DAC_ADC_raw	out_voltage6_mode_6_DAC_ADC_raw	

Connect port2 (DAC) to port0 (ADC).

Write to the port2 (DAC):

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 500 > out_voltage2_mode_5_DAC_raw
```

```
max11300 spi0.0: the DAC data register is 62
```

```
max11300 spi0.0: the value in the DAC data register is 1f4
```

Read the port0 (ADC):

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_voltage0_mode_7_ADC_raw
```

```
max11300 spi0.0: read SE channel
```

```
499
```

Connect port2 (DAC) to port4 (ADC differential positive)
and port3 (DAC) to port 5 (ADC differential negative).

Set 5V output in the port2 (DAC):

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 2047 > out_voltage2_mode_5_DAC_raw
max11300 spi0.0: the DAC data register is 62
max11300 spi0.0: the value in the DAC data register is 7ff
```

Set 2.5V in the port3 (DAC):

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 1024 > out_voltage2_mode_5_DAC_raw
max11300 spi0.0: the DAC data register is 62
max11300 spi0.0: the value in the DAC data register is 400
```

Read differential input (port4_port5). The read value will be 2.5V aprox:

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_voltage4-voltage5_mode_8_ADC_raw
max11300 spi0.0: read differential channel
512
```

Write DAC and read ADC in the port6:

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 1024 > out_voltage6_mode_6_DAC_ADC_raw
max11300 spi0.0: the DAC data register is 66
max11300 spi0.0: the value in the DAC data register is 400
```

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_voltage6_mode_6_DAC_ADC_raw
max11300 spi0.0: read SE channel
1022
```

See the gpio chip controllers. The gpio controller of the MAX11300 device is called gpiochip3:

```
root@raspberrypi:/home/pi# ls -l /dev/gpiochip*
crw-rw---- 1 root gpio 254, 0 dic  5 12:52 /dev/gpiochip0
crw-rw---- 1 root gpio 254, 1 dic  5 12:52 /dev/gpiochip1
crw-rw---- 1 root gpio 254, 2 dic  5 12:52 /dev/gpiochip2
crw-rw---- 1 root gpio 254, 3 dic  5 12:53 /dev/gpiochip3
```

Print information of all the lines of the gpiochip3:

```
root@raspberrypi:/home/pi# gpioinfo gpiochip3
gpiochip3 - 4 lines:
    line 0:      unnamed      unused   input   active-high
    line 1:      unnamed      unused   input   active-high
    line 2:      unnamed      unused   input   active-high
    line 3:      unnamed      unused   input   active-high
```

Connect port19 (GPO) to port 18 (GPI).

Set port19 (GPO) to high:

```
root@raspberrypi:/home/pi# gpioset gpiochip3 3=1
max11300 spi0.0: The GPIO is set as an output
max11300 spi0.0: The GPIO output is set
max11300 spi0.0: The GPIO output is set high and port_number is 19. Pin is > 0x0F
```

Read port 18 (GPI):

```
root@raspberrypi:/home/pi# gpioget gpiochip3 2
```

```
max11300 spi0.0: The GPIO is set as an input
max11300 spi0.0: The GPIO input is get
max11300 spi0.0: read SE channel
1
```

Set port19 (GPO) to low:

```
root@raspberrypi:/home/pi# gpiowrite gpiochip3 3=0
max11300 spi0.0: The GPIO is set as an output
max11300 spi0.0: The GPIO output is set
max11300 spi0.0: The GPIO output is set low and port_number is 19. Pin is > 0x0F
```

Read port 18 (GPI):

```
root@raspberrypi:/home/pi# gpioread gpiochip3 2
max11300 spi0.0: The GPIO is set as an input
max11300 spi0.0: The GPIO input is get
max11300 spi0.0: read SE channel
0
```

Connect port19 (GPO) to port 7 (GPI).

Set port19 (GPO) to high:

```
root@raspberrypi:/home/pi# gpiowrite gpiochip3 3=1
max11300 spi0.0: The GPIO is set as an output
max11300 spi0.0: The GPIO output is set
max11300 spi0.0: The GPIO output is set high and port_number is 19. Pin is > 0x0F
```

Read port7 (GPI):

```
root@raspberrypi:/home/pi# gpioread gpiochip3 0
max11300 spi0.0: The GPIO is set as an input
max11300 spi0.0: The GPIO input is get
max11300 spi0.0: read SE channel
1
```

Set port19 (GPO) to low:

```
root@raspberrypi:/home/pi# gpiowrite gpiochip3 3=0
max11300 spi0.0: The GPIO is set as an output
max11300 spi0.0: The GPIO output is set
max11300 spi0.0: The GPIO output is set low and port_number is 19. Pin is > 0x0F
```

Read port7 (GPI):

```
root@raspberrypi:/home/pi# gpioread gpiochip3 0
max11300 spi0.0: The GPIO is set as an input
max11300 spi0.0: The GPIO input is get
max11300 spi0.0: read SE channel
0
```

Remove the modules:

```
root@raspberrypi:/home/pi# rmmod max11300.ko
root@raspberrypi:/home/pi# rmmod max11300-base.ko
```


In this section, you have seen how to control the GPIOs by using the tools provided with libgpiod. In the next section, you will see how to write applications that control the GPIOs by using two different methods. The first method will control a GPIO by using a device node, and the second method will control a GPIO by using the functions of the libgpiod library.

GPIO control through a character device

In the Chapter 5, you learned how to write drivers that use the GPIO descriptor interface included in the GPIOlib framework to control GPIOs. This descriptor interface identifies each GPIO through a `gpio_desc` structure.

GPIOlib is a framework that provides an internal Linux kernel API for managing and configuring GPIOs, acting as a bridge between the Linux GPIO controller drivers and the Linux GPIO client drivers. Writing Linux drivers for controlling the GPIOs of a device is a good practice, but you can prefer to control the GPIOs from user space. GPIOlib also provides access to the user space APIs that control the GPIOs through `ioctl` calls on `/dev/gpiochipX` char device files, where `X` is the number of the GPIO bank.

Until the launching of Linux kernel 4.8, the GPIOs were accessed using the `sysfs` (`/sys/class/gpio`) method, but after this release, there are new interfaces based on a char device. The `sysfs` interface is deprecated and is highly recommended to use the new interface. These are some of the advantages of using the new character device user API:

- One device file for each `gpiochip`: `/dev/gpiochip0`, `/dev/gpiochip1`, `/dev/gpiochipX...`
- Similar to other kernel interfaces: `ioctl()` + `poll()` + `read()` + `close()`
- Possible to set/read multiple GPIOs at once.
- Possible to find GPIO lines by name.

The following application toggles ten times the `port19` of the `PIXI™ CLICK` board. The `port19` can be connected to the red LED on the Color click eval board (<https://www.mikroe.com/color-click>) to see the red LED blinking.

Create a new `application_code` folder inside the `linux_5.4_max11300_driver` folder:

```
~/linux_5.4_rpi3_drivers/linux_5.4_max11300_driver$ mkdir application_code
```

Create a new `gpio_device_app.c` file in the `application_code` folder, send the application to the Raspberry Pi, and compile the application on the Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_max11300_driver/application_code$ scp gpio_device_app.c
root@10.0.0.10:/home/pi
root@raspberrypi:/home/pi# gcc -o gpio_device_app gpio_device_app.c
```

Finally, execute the application on the target. You can see the red LED flashes!

Load the modules:

```
root@raspberrypi:/home/pi# insmod max11300-base.ko
```

```
root@raspberrypi:/home/pi# insmod max11300.ko
```

Execute the application:

```
root@raspberrypi:/home/pi# ./gpio_device_app
```

Listing 11-5: gpio_device_app.c

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <linux/gpio.h>
#include <sys/ioctl.h>

/* Configure port19 as an output, and flash an LED */

#define DEVICE_GPIO "/dev/gpiochip3"

int main(int argc, char *argv[])
{
    int fd;
    int ret;
    int flash = 10;

    struct gpiohandle_data data;
    struct gpiohandle_request req;

    /* Open gpio device */
    fd = open(DEVICE_GPIO, 3);
    if (fd < 0) {
        fprintf(stderr, "Failed to open %s\n", DEVICE_GPIO);
        return -1;
    }

    /* Request GPIO line 3 as an output (red LED) */
    req.lineoffsets[0] = 3;
    req.lines = 1;
    req.flags = GPIOHANDLE_REQUEST_OUTPUT;
    strcpy(req.consumer_label, "led_gpio_port19");

    ret = ioctl(fd, GPIO_GET_LINEHANDLE_IOCTL, &req);
    if (ret < 0) {
        printf("ERROR get line handle IOCTL (%d)\n", ret);
        if (close(fd) == -1)
            perror("Failed to close GPIO char device");
    }
}
```

```

        return ret;
    }

    /* Start the led_red with off state */
    data.values[0] = 1;

    for (int i=0; i < flash; i++) {
        /* toggle LED */
        data.values[0] = !data.values[0];
        ret = ioctl(req.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL, &data);
        if (ret < 0) {
            fprintf(stderr, "Failed to issue %s (%d)\n",
                    "GPIOHANDLE_SET_LINE_VALUES_IOCTL", ret);
            if (close(req.fd) == -1)
                perror("Failed to close GPIO line");
            if (close(fd) == -1)
                perror("Failed to close GPIO char device");
            return ret;
        }
        sleep(1);
    }

    /* Close gpio line */
    ret = close(req.fd);
    if (ret == -1)
        perror("Failed to close GPIO line");

    /* Close gpio device */
    ret = close(fd);
    if (ret == -1)
        perror("Failed to close GPIO char device");

    return ret;
}

```

GPIO control through the gpiolibd library

In this section, you will see how to control GPIOs using the functions of the libgpiod library.

The following libgpiod_app application has the same behaviour than the gpio_device_app one, toggling ten times the port19 connected to the red LED on the Color click eval board, but this time, you will use the libgpiod library instead of the "gpio char device" method to control the red LED.

Create a new libgpiod_max11300_app.c file in the application_code folder, and send the application to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_max11300_driver/application_code$ scp libgpiod_max11300_app.c root@10.0.0.10:/home/pi
```

Compile the application on the Raspberry Pi:

```
root@raspberrypi:/home/pi# gcc -o libgpiod_max11300_app -lgpiod libgpiod_max11300_app.c
```

Finally, execute the compiled application on the target. You can see the red LED flashes!:

Load the modules:

```
root@raspberrypi:/home/pi# insmod max11300-base.ko
```

```
root@raspberrypi:/home/pi# insmod max11300.ko
```

Execute the application:

```
root@raspberrypi:/home/pi# ./libgpiod_max11300_app
```

Listing 11-6: libgpiod_max11300_app.c

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <gpiod.h>

int main(int argc, char *argv[])
{
    struct gpiod_chip *output_chip;
    struct gpiod_line *output_line;
    int line_value = 1;
    int flash = 10;
    int ret;

    /* Open /dev/gpiochip3 */
    output_chip = gpiod_chip_open_by_number(3);
    if (!output_chip)
        return -1;

    /* Get line 3 (port19) of the gpiochip3 device */
    output_line = gpiod_chip_get_line(output_chip, 3);
    if(!output_line) {
        gpiod_chip_close(output_chip);
        return -1;
    }

    /* Config port19 (GPO) as output and set output to high level */
    if (gpiod_line_request_output(output_line, "Port19_GPO",
                                   GPIO_LINE_ACTIVE_STATE_HIGH) == -1) {
        gpiod_line_release(output_line);
        gpiod_chip_close(output_chip);
        return -1;
    }

    /* Toggle 10 times the port19 (GPO) of the max11300 device */
    for (int i=0; i < flash; i++) {
        line_value = !line_value;
        ret = gpiod_line_set_value(output_line, line_value);
        if (ret == -1) {
            ret = -errno;
            gpiod_line_release(output_line);
        }
    }
}
```

```

        gpiod_chip_close(output_chip);
        return ret;
    }
    sleep(1);
}

gpiod_line_release(output_line);
gpiod_chip_close(output_chip);

return 0;
}

```

LAB 11.2: “Nunchuk provider and consumer” modules

The IIO drivers can export their channels to other consumer drivers that can use them. These devices are typically ADCs that will have other consumers within the kernel; for example, battery chargers, fuel gauges and even thermal sensors can use the IIO ADC driver’s channels. In this lab, you are going to develop two drivers: the first one is an IIO provider driver which will read the 3-axis data from the Nunchuk’s accelerometer sensor; the second one is an Input subsystem consumer driver which will read the IIO channel values from the IIO provider driver and report them to the Input subsystem.

Accelerometers are usually used in consumer equipments, and their availability over time is very limited. Using this provider-consumer approach, it will be easy to replace the accelerometer of the Nunchuk controller with a new one that has an IIO driver available in the kernel mainline (or you can develop your own) and reuse the consumer driver without doing any modification.

You will use the same HW setup of the previous LAB 10.2.

Nunchuk provider module

Let's start with the development of the Nunchuk provider driver. The operation of the driver will be quite simple. You will use the IIO subsystem to develop a driver that will read the acceleration of the Nunchuk’s accelerometer axes.

The main code sections of the driver will now be described:

1. Include the function headers:

```

#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>

```

2. Create a private nunchuk_accel structure:

```

struct nunchuk_accel {
    struct i2c_client *client;

```

```
};
```

3. In the `nunchuk_accel_probe()` function, declare an instance of the private structure, and allocate the `iio_dev` structure:

```
struct iio_dev *indio_dev;
struct nunchuk_accel *nunchuk_accel;
indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*nunchuk_accel));
```

4. Initialize the `iio_device` and the `nunchuk_accel` private structure within the `nunchuk_accel_probe()` function. The `nunchuk_accel` private structure will be previously allocated by using the `iio_priv()` function. Keep pointers between physical devices (devices as handled by the physical bus, I2C in this case) and logical devices:

```
nunchuk_accel = iio_priv(indio_dev); /* To be able to access the private structure in
other functions of the driver you need to attach it to the iio_dev structure using
the iio_priv() function. You will retrieve the pointer "nunchuk_accel" to the private
structure using the same function iio_priv() */

nunchuk_accel->client = client; /* Keep pointer to the I2C device, needed for exchanging
data with the Nunchuk device */

indio_dev->name = "Nunchuk Accel"; /* Store the iio_dev name */

indio_dev->dev.parent = &client->dev; /* keep pointers between physical devices (devices
as handled by the physical bus, I2C in this case) and logical devices */

indio_dev->info = &nunchuk_info; /* Store the address of the iio_info structure, which
contains a pointer variable to the IIO raw reading callback */

indio_dev->channels = nunchuk_channels; /* Store address of the iio_chan_spec array of
structures */

indio_dev->num_channels = 3; /* Set number of channels of the device */

indio_dev->modes = INDIO_DIRECT_MODE; /* Device operating mode. DIRECT_MODE indicates
that the collected data is not cached, and the single data can be read directly under
sysfs */
```

5. Register the device to the IIO core. Now, the device is global to the rest of the driver functions until it is unregistered. After this call, the device is ready to accept requests from user space applications.

```
devm_iio_device_register(&client->dev, indio_dev);
```

6. Create the `iio_chan_spec` structures to expose to user space the `sysfs` attributes of each channel. The type variable specifies what type of measurement the channel makes, acceleration in the case of our driver. The modified field of `iio_chan_spec` is set to 1. Modifiers are specified by using the `channel2` field of the same struct `iio_chan_spec` and are used to indicate a physically unique characteristic of the channel, such as the acceleration axis in the case of our driver. The `info_mask_separate` variable indicates which information should be unique to the channel.

```

#define NUNCHUK_IIO_CHAN(axis) {
    .type = IIO_ACCEL,
    .modified = 1,
    .channel2 = IIO_MOD_##axis,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
}

static const struct iio_chan_spec nunchuk_channels[] = {
    NUNCHUK_IIO_CHAN(X),
    NUNCHUK_IIO_CHAN(Y),
    NUNCHUK_IIO_CHAN(Z),
};

```

The IIO channel definitions above will generate the following data channel access attributes below:

```

/sys/bus/iio/devices/iio:deviceX/in_accel_x_raw
/sys/bus/iio/devices/iio:deviceX/in_accel_y_raw
/sys/bus/iio/devices/iio:deviceX/in_accel_z_raw

```

7. Create the `iio_info` structure to declare the hooks that the IIO core will use for this device. There is only one kernel hook available for interactions from user space with the `sysfs` attributes of each channel.

```

static const struct iio_info nunchuk_info = {
    .read_raw = nunchuk_accel_read_raw,
};

```

The `nunchuk_accel_read_raw()` function returns the value of each axis when any of the `in_accel_X_raw` entries is read from user space. The axis of the accelerometer will be filtered using the `channel2` modifier.

```

static int nunchuk_accel_read_raw(struct iio_dev *indio_dev,
                                struct iio_chan_spec const *chan,
                                int *val, int *val2, long mask)
{
    char buf[6];
    struct nunchuk_accel *nunchuk_accel = iio_priv(indio_dev);
    struct i2c_client *client = nunchuk_accel->client;

    /* Read data from nunchuk */
    nunchuk_read_registers(client, buf, ARRAY_SIZE(buf));

    /* Data needs to be written to 'val' and 'val2' is ignored */
    switch (chan->channel2) {
    case IIO_MOD_X:
        *val = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
        break;
    case IIO_MOD_Y:
        *val = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
        break;
    case IIO_MOD_Z:
        *val = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);

```

```

        break;
    default:
        return -EINVAL;
    }

    return IIO_VAL_INT;
}

```

8. Declare a list of devices supported by the driver:

```

static const struct of_device_id nunchuk_accel_of_match[] = {
    { .compatible = "nunchuk_accel"},
    {}
};
MODULE_DEVICE_TABLE(of, nunchuk_accel_of_match);

```

9. Define an array of `i2c_device_id` structures:

```

static const struct i2c_device_id nunchuk_accel_id[] = {
    { "nunchuk_accel", 0 },
    {}
};
MODULE_DEVICE_TABLE(i2c, nunchuk_accel_id);

```

10. Add an `i2c_driver` structure that will be registered to the I2C bus:

```

static struct i2c_driver nunchuk_accel_driver = {
    .driver = {
        .name = "nunchuk_accel",
        .owner = THIS_MODULE,
        .of_match_table = nunchuk_accel_of_match,
    },
    .probe = nunchuk_accel_probe,
    .remove = nunchuk_accel_remove,
    .id_table = nunchuk_accel_id,
};

```

11. Register your driver with the I2C bus:

```

module_i2c_driver(nunchuk_accel_driver);

```


Listing 11-7: nunchuk_accel.c

```

#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>
#include <linux/iio/iio.h>

struct nunchuk_accel {
    struct i2c_client *client;
};

#define NUNCHUK_IIO_CHAN(axis) {
    .type = IIO_ACCEL,
    .modified = 1,
    .channel2 = IIO_MOD_##axis,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
}

static const struct iio_chan_spec nunchuk_channels[] = {
    NUNCHUK_IIO_CHAN(X),
    NUNCHUK_IIO_CHAN(Y),
    NUNCHUK_IIO_CHAN(Z),
};

static int nunchuk_read_registers(struct i2c_client *client, char *buf, int buf_size)
{
    int ret;

    mdelay(10);

    buf[0] = 0x00;
    ret = i2c_master_send(client, buf, 1);
    if (ret >= 0 && ret != 1)
        return -EIO;
    if (ret < 0)
        return ret;

    mdelay(10);

    ret = i2c_master_recv(client, buf, buf_size);
    if (ret >= 0 && ret != buf_size)
        return -EIO;
    if (ret < 0)
        return ret;

    return 0;
}

static int nunchuk_accel_read_raw(struct iio_dev *indio_dev,
                                struct iio_chan_spec const *chan,
                                int *val, int *val2, long mask)
{
    char buf[6];

```

```

struct nunchuk_accel *nunchuk_accel = iio_priv(indio_dev);
struct i2c_client *client = nunchuk_accel->client;

/* Read data from nunchuk */
if (nunchuk_read_registers(client, buf, ARRAY_SIZE(buf)) < 0)
{
    dev_info(&client->dev, "Error reading the nunchuk registers.\n");
    return -EIO;
}

/* Data needs to be written to 'val' and 'val2' is ignored */
switch (chan->channel2) {
case IIO_MOD_X:
    *val = (buf[2] << 2) | ((buf[5] >> 2) & 0x3);
    break;
case IIO_MOD_Y:
    *val = (buf[3] << 2) | ((buf[5] >> 4) & 0x3);
    break;
case IIO_MOD_Z:
    *val = (buf[4] << 2) | ((buf[5] >> 6) & 0x3);
    break;
default:
    return -EINVAL;
}

return IIO_VAL_INT;
}

static const struct iio_info nunchuk_info = {
    .read_raw      = nunchuk_accel_read_raw,
};

static int nunchuk_accel_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    int ret;
    u8 buf[2];
    struct iio_dev *indio_dev;

    /* Declare an instance of the private structure */
    struct nunchuk_accel *nunchuk_accel;

    dev_info(&client->dev, "nunchuk_accel_probe() function is called.\n");

    /* Allocate the iio_dev structure */
    indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*nunchuk_accel));
    if (indio_dev == NULL)
        return -ENOMEM;

    nunchuk_accel = iio_priv(indio_dev);
    /* Associate client->dev with nunchuk private structure */
    i2c_set_clientdata(client, nunchuk_accel);
    nunchuk_accel->client = client;

    indio_dev->name = "Nunchuk Accel";

```

```

indio_dev->dev.parent = &client->dev;
indio_dev->info = &nunchuk_info;
indio_dev->channels = nunchuk_channels;
indio_dev->num_channels = 3;
indio_dev->modes = INDIO_DIRECT_MODE;

/* Nunchuk handshake */
buf[0] = 0xf0;
buf[1] = 0x55;
ret = i2c_master_send(client, buf, 2);
if (ret >= 0 && ret != 2)
    return -EIO;
if (ret < 0)
    return ret;

udelay(1);

buf[0] = 0xfb;
buf[1] = 0x00;
ret = i2c_master_send(client, buf, 1);
if (ret >= 0 && ret != 1)
    return -EIO;
if (ret < 0)
    return ret;

ret = devm_iio_device_register(&client->dev, indio_dev);
if (ret)
    return ret;

dev_info(&client->dev, "nunchuk registered\n");

return 0;
}

static int nunchuk_accel_remove(struct i2c_client *client)
{
    dev_info(&client->dev, "nunchuk_remove()\n");
    return 0;
}

/* Add entries to the Device Tree */
static const struct of_device_id nunchuk_accel_of_match[] = {
    { .compatible = "nunchuk_accel"},
    {}
};
MODULE_DEVICE_TABLE(of, nunchuk_accel_of_match);

static const struct i2c_device_id nunchuk_accel_id[] = {
    { "nunchuk_accel", 0 },
    {}
};
MODULE_DEVICE_TABLE(i2c, nunchuk_accel_id);

```

```

/* Create struct i2c_driver */
static struct i2c_driver nunchuk_accel_driver = {
    .driver = {
        .name = "nunchuk_accel",
        .owner = THIS_MODULE,
        .of_match_table = nunchuk_accel_of_match,
    },
    .probe = nunchuk_accel_probe,
    .remove = nunchuk_accel_remove,
    .id_table = nunchuk_accel_id,
};

/* Register to I2C bus as a driver */
module_i2c_driver(nunchuk_accel_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a nunchuk Accelerometer IIO framework I2C driver");

```

Nunchuk consumer module

Let's continue with the development of the Nunchuk consumer driver. You will use the Input subsystem to develop a platform driver that will consume the IIO channels of the Nunchuk provider driver.

The Nunchuk consumer driver will be very similar to the Nunchuk driver which was developed in the LAB 10.2, so we will only describe the code used for the consumer operations:

1. You will declare a private structure which includes pointers variables to the iio_channels structures:

```

/* Create private structure */
struct nunchuk_dev {
    struct input_polled_dev *polled_input;
    /* declare pointers to the IIO channels of the provider device */
    struct iio_channel *accel_x, *accel_y, *accel_z;
};

```

2. In the nunchuk_probe() function, you will get the pointers to the iio_channels of the provider device and store them in your private structure:

```

/* Get pointer to channel "accel_x" of the provider device */
nunchuk->accel_x = devm_iio_channel_get(dev, "accel_x");

/* Get pointer to channel "accel_y" of the provider device */
nunchuk->accel_y = devm_iio_channel_get(dev, "accel_y");

/* Get pointer to channel "accel_z" of the provider device */
nunchuk->accel_z = devm_iio_channel_get(dev, "accel_z");

```

3. In the nunchuk_poll() function, you will get the IIO channel raw values from the provider device and report the ABS_RX, ABS_RY and ABS_RZ events to the Input subsystem:

```

static void nunchuk_poll(struct input_polled_dev *polled_input)
{
    int accel_x, accel_y, accel_z;
    struct nunchuk_dev *nunchuk;

    nunchuk = polled_input->private;

    /* Read IIO "accel_x" channel raw value from the provider device */
    iio_read_channel_raw(nunchuk->accel_x, &accel_x);

    /* Report ABS_RX event to the input system */
    input_report_abs(polled_input->input, ABS_RX, accel_x);

    /* Read IIO "accel_y" channel raw value from the provider device */
    iio_read_channel_raw(nunchuk->accel_y, &accel_y);

    /* Report ABS_RY event to the input system */
    input_report_abs(polled_input->input, ABS_RY, accel_y);

    /* Read IIO "accel_z" channel raw value from the provider device */
    iio_read_channel_raw(nunchuk->accel_z, &accel_z);

    /* Report ABS_RZ event to the input system */
    input_report_abs(polled_input->input, ABS_RZ, accel_z);

    input_sync(polled_input->input);
}

```

Listing 11-8: nunchuk_consumer.c

```

#include <linux/module.h>
#include <linux/delay.h>
#include <linux/input.h>
#include <linux/iio/consumer.h>
#include <linux/iio/types.h>
#include <linux/input-polldev.h>
#include <linux/platform_device.h>

/* Create private structure */
struct nunchuk_dev {
    struct input_polled_dev *polled_input;
    /* declare pointers to the IIO channels of the provider device */
    struct iio_channel *accel_x, *accel_y, *accel_z;
};

/*
 * The poll handler function reads the hardware,
 * queues events to be reported (input_report_*)
 * and flushes the queued events (input_sync)
 */
static void nunchuk_poll(struct input_polled_dev *polled_input)
{

```

```

int accel_x, accel_y, accel_z;
struct nunchuk_dev *nunchuk;
int ret;

nunchuk = polled_input->private;

/* Read IIO "accel_x" channel raw value from the provider device */
ret = iio_read_channel_raw(nunchuk->accel_x, &accel_x);
if (unlikely(ret < 0))
    return;

/* Report ABS_RX event to the input system */
input_report_abs(polled_input->input, ABS_RX, accel_x);

/* Read IIO "accel_y" channel raw value from the provider device */
ret = iio_read_channel_raw(nunchuk->accel_y, &accel_y);
if (unlikely(ret < 0))
    return;

/* Report ABS_RY event to the input system */
input_report_abs(polled_input->input, ABS_RY, accel_y);

/* Read IIO "accel_z" channel raw value from the provider device */
ret = iio_read_channel_raw(nunchuk->accel_z, &accel_z);
if (unlikely(ret < 0))
    return;

/* Report ABS_RZ event to the input system */
input_report_abs(polled_input->input, ABS_RZ, accel_z);

/*
 * Tell those who receive the events
 * that a complete report has been sent
 */
input_sync(polled_input->input);
}

static int nunchuk_probe(struct platform_device *pdev)
{
    int ret;
    struct device *dev = &pdev->dev;
    enum iio_chan_type type;

    /* Declare a pointer to the private structure */
    struct nunchuk_dev *nunchuk;

    /* Declare pointers to input_dev and input_polled_dev structures */
    struct input_polled_dev *polled_device;
    struct input_dev *input;

    dev_info(dev, "nunchuk_probe() function is called.\n");

    /* Allocate private structure for nunchuk device */
    nunchuk = devm_kzalloc(dev, sizeof(*nunchuk), GFP_KERNEL);

```

```
if (nunchuk == NULL)
    return -ENOMEM;

/* Get pointer to channel "accel_x" of the provider device */
nunchuk->accel_x = devm_iio_channel_get(dev, "accel_x");
if (IS_ERR(nunchuk->accel_x))
    return PTR_ERR(nunchuk->accel_x);

if (!nunchuk->accel_x->indio_dev)
    return -ENXIO;

/* Get type of "accel_x" channel */
ret = iio_get_channel_type(nunchuk->accel_x, &type);
if (ret < 0)
    return ret;

if (type != IIO_ACCEL) {
    dev_err(dev, "not accelerometer channel %d\n", type);
    return -EINVAL;
}

/* Get pointer to channel "accel_y" of the provider device */
nunchuk->accel_y = devm_iio_channel_get(dev, "accel_y");
if (IS_ERR(nunchuk->accel_y))
    return PTR_ERR(nunchuk->accel_y);

if (!nunchuk->accel_y->indio_dev)
    return -ENXIO;

/* Get type of "accel_y" channel */
ret = iio_get_channel_type(nunchuk->accel_y, &type);
if (ret < 0)
    return ret;

if (type != IIO_ACCEL) {
    dev_err(dev, "not accel channel %d\n", type);
    return -EINVAL;
}

/* Get pointer to channel "accel_z" of the provider device */
nunchuk->accel_z = devm_iio_channel_get(dev, "accel_z");
if (IS_ERR(nunchuk->accel_z))
    return PTR_ERR(nunchuk->accel_z);

if (!nunchuk->accel_z->indio_dev)
    return -ENXIO;

/* Get type of "accel_z" channel */
ret = iio_get_channel_type(nunchuk->accel_z, &type);
if (ret < 0)
    return ret;

if (type != IIO_ACCEL) {
    dev_err(dev, "not accel channel %d\n", type);
```

```

        return -EINVAL;
    }

    /* Allocate struct input_polled_dev */
    polled_device = devm_input_allocate_polled_device(dev);
    if (!polled_device) {
        dev_err(dev, "unable to allocate input device\n");
        return -ENOMEM;
    }

    /* Initialize the polled input device */
    /* To recover nunchuk in the poll() function */
    polled_device->private = nunchuk;

    /* Fill in the poll interval */
    polled_device->poll_interval = 50;

    /* Fill in the poll handler */
    polled_device->poll = nunchuk_poll;

    polled_device->input->name = "WII accel consumer";
    polled_device->input->id.bustype = BUS_HOST;

    /*
     * Store the polled device in the global structure
     * to recover it in the remove() function
     */
    nunchuk->polled_input = polled_device;

    input = polled_device->input;

    /* To recover nunchuck structure from remove() function */
    platform_set_drvdata(pdev, nunchuk);

    /*
     * Set EV_ABS type events and from those
     * ABS_X, ABS_Y, ABS_RX, ABS_RY and ABS_RZ event codes
     */
    set_bit(EV_ABS, input->evbit);
    set_bit(ABS_RX, input->absbit); /* accelerometer */
    set_bit(ABS_RY, input->absbit);
    set_bit(ABS_RZ, input->absbit);

    /*
     * Fill additional fields in the input_dev struct for
     * each absolute axis nunchuk has
     */
    input_set_abs_params(input, ABS_RX, 0x00, 0x3ff, 0, 0);
    input_set_abs_params(input, ABS_RY, 0x00, 0x3ff, 0, 0);
    input_set_abs_params(input, ABS_RZ, 0x00, 0x3ff, 0, 0);

    /* Finally, register the input device */
    ret = input_register_polled_device(nunchuk->polled_input);
    if (ret < 0)

```



```

        return ret;

    return 0;
}

static int nunchuk_remove(struct platform_device *pdev)
{
    struct nunchuk_dev *nunchuk = platform_get_drvdata(pdev);
    input_unregister_polled_device(nunchuk->polled_input);
    dev_info(&pdev->dev, "nunchuk_remove()\n");

    return 0;
}

/* Add entries to the Device Tree */
/* Declare a list of devices supported by the driver */
static const struct of_device_id nunchuk_of_ids[] = {
    { .compatible = "nunchuk_consumer",
      {} },
};
MODULE_DEVICE_TABLE(of, nunchuk_of_ids);

/* Define a platform driver structure */
static struct platform_driver nunchuk_platform_driver = {
    .probe = nunchuk_probe,
    .remove = nunchuk_remove,
    .driver = {
        .name = "nunchuk_consumer",
        .of_match_table = nunchuk_of_ids,
        .owner = THIS_MODULE,
    }
};

/* Register the platform driver */
module_platform_driver(nunchuk_platform_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a Nunchuk consumer platform driver");

```

LAB 11.2 Device Tree description

Open the `bcm2710-rpi-3-b.dts` DT file, and find the `i2c1` controller master node. In the `i2c1` node, you can see a `pinctrl-0` property that points to the `i2c1_pins` pin configuration node, which configures the pins of the `i2c1` controller in I2C mode.

The `i2c1` controller is enabled by writing "okay" to the status property. You will set to 100Khz the clock-frequency property. The Nunchuk device communicates with the Raspberry Pi using an I2C bus interface with a maximum frequency of 100kHz.

Now, you will remove or comment out the `nunchuk` node coming from the LAB 10.3 and add the `nunchuk_accel` node of our provider driver to the `i2c1` controller node.

You can see below in bold the Device Tree configuration for our `nunchuk_accel` device. The `io-channel-cells` property provides the number of cells in an IIO specifier, typically 0 for nodes with a single IIO output and 1 for nodes with multiple IIO outputs (as it is the case of our Nunchuk provider driver).

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;
    status = "okay";

    nunchuk_accel: nunchuk_accel@52 {
        #io-channel-cells = <1>;
        compatible = "nunchuk_accel";
        reg = <0x52>;
    };

    /* nunchuk: nunchuk@52 {
        compatible = "nunchuk";
        reg = <0x52>;
    }; */

};
```

Now, add the `nunchuk_consumer` node of our consumer driver to the `soc` node. The `io-channels` property provides a list of phandle and IIO specifier pairs, one pair for each IIO input to the device. The `io-channel-names` property provides a list of IIO input name strings sorted in the same order as the `io-channels` property. Consumer drivers will use the `io-channel-names` property to match IIO input names with IIO specifiers.

```
&soc {
    virtgpio: virtgpio {
        compatible = "brcm,bcm2835-virtgpio";
        gpio-controller;
        #gpio-cells = <2>;
        firmware = <&firmware>;
        status = "okay";
    };

    nunchuk_consumer {
        compatible = "nunchuk_consumer";
        io-channels = <&nunchuk_accel 0>, <&nunchuk_accel 1>, <&nunchuk_accel2>;
        io-channel-names = "accel_x", "accel_y", "accel_z";
    };
};
```

LAB 11.2 driver demonstration

Create the `nunchuk_accel.c` and `nunchuk_consumer.c` files in the `nunchuk_drivers` folder, and add `nunchuk_accel.o` and `nunchuk_consumer.o` to your Makefile `obj-m` variable, then build and deploy the modules to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/nunchuk_drivers$ make
~/linux_5.4_rpi3_drivers/nunchuk_drivers$ make deploy
```

Build the modified Device Tree, and load it to the target processor:

```
~/linux_rpi3/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
~/linux_rpi3/linux$ scp arch/arm/boot/dts/bcm2710-rpi-3-b.dtb root@10.0.0.10:/boot/
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Follow the next instructions to test the driver:

Load the provider module:

```
root@raspberrypi:/home/pi# insmod nunchuk_accel.ko
nunchuk_accel 1-0052: nunchuk_accel_probe() function is called.
nunchuk_accel 1-0052: nunchuk registered
```

See the sysfs entries under the nunchuk iio device:

```
root@raspberrypi:/home/pi# cd /sys/bus/iio/devices/iio\:device0
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat name
Nunchuk Accel
```

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# ls
dev          in_accel_y_raw  name           power          uevent
in_accel_x_raw in_accel_z_raw  of_node        subsystem
```

Play with the nunchuk and read the value of the accel axes:

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_x_raw
1007
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_x_raw
856
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_x_raw
703
```

Load the consumer driver:

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cd /home/pi/
root@raspberrypi:/home/pi# insmod nunchuk_consumer.ko
```

Launch the evtest application and play with the nunchuk device:

```
root@raspberrypi:/home/pi# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0:      WII accel consumer
Select the device event number [0-0]: 0
Input driver version is 1.0.1
```

Input device ID: bus 0x19 vendor 0x0 product 0x0 version 0x0

Input device name: "WII accel consumer"

Supported events:

Event type 0 (EV_SYN)

Event type 3 (EV_ABS)

Event code 3 (ABS_RX)

Value 650

Min 0

Max 1023

Event code 4 (ABS_RY)

Value 500

Min 0

Max 1023

Event code 5 (ABS_RZ)

Value 582

Min 0

Max 1023

Properties:

Testing ... (interrupt to exit)

Event: time 1608593843.374146, type 3 (EV_ABS), code 3 (ABS_RX), value 650

Event: time 1608593843.374146, type 3 (EV_ABS), code 4 (ABS_RY), value 500

Event: time 1608593843.374146, type 3 (EV_ABS), code 5 (ABS_RZ), value 582

Event: time 1608593843.374146, ----- SYN_REPORT -----

Event: time 1608593843.493549, type 3 (EV_ABS), code 3 (ABS_RX), value 499

Event: time 1608593843.493549, type 3 (EV_ABS), code 4 (ABS_RY), value 374

Event: time 1608593843.493549, type 3 (EV_ABS), code 5 (ABS_RZ), value 421

Event: time 1608593843.493549, ----- SYN_REPORT -----

Event: time 1608593843.613407, type 3 (EV_ABS), code 3 (ABS_RX), value 425

Event: time 1608593843.613407, type 3 (EV_ABS), code 4 (ABS_RY), value 379

Event: time 1608593843.613407, type 3 (EV_ABS), code 5 (ABS_RZ), value 438

Event: time 1608593843.613407, ----- SYN_REPORT -----

Event: time 1608593843.733405, type 3 (EV_ABS), code 3 (ABS_RX), value 452

Event: time 1608593843.733405, type 3 (EV_ABS), code 4 (ABS_RY), value 340

Event: time 1608593843.733405, type 3 (EV_ABS), code 5 (ABS_RZ), value 233

Event: time 1608593843.733405, ----- SYN_REPORT -----

Event: time 1608593843.853410, type 3 (EV_ABS), code 3 (ABS_RX), value 158

Event: time 1608593843.853410, type 3 (EV_ABS), code 4 (ABS_RY), value 469

Event: time 1608593843.853410, type 3 (EV_ABS), code 5 (ABS_RZ), value 517

Press ^C to exit:

root@raspberrypi:/home/pi#

Remove the consumer module:

root@raspberrypi:/home/pi# rmmod nunchuk_consumer.ko

Remove the provider module:

root@raspberrypi:/home/pi# rmmod nunchuk_accel.ko

12

Using the Regmap API in Device Drivers

As you have seen throughout the different chapters of this book, Linux has subsystems such as I2C and SPI which are used to connect to devices that reside on these buses. Both these buses have the common function of reading and writing registers from the devices connected to them. This often causes redundant code to be present in the subsystems that have this register read and write functionality.

To avoid this and to factor out common code, as well as for easy driver maintenance and development, Linux developers introduced a new kernel API from version 3.1, which is called **regmap**. This infrastructure was previously present in the Linux ASoC (ALSA) subsystem, but has now been made available for the entire Linux through the regmap API (defined in `include/linux/regmap.h` and implemented in `drivers/base/regmap/` in the kernel source tree). The regmap subsystem abstracts the access to the registers of SPI and I2C devices and also to memory-mapped registers (MMIO). The regmap subsystem also provides a caching mechanism that can reduce the number of accesses to the device and can handle IRQ chips and IRQs.

So far, you have developed several I2C and SPI device drivers by using the specific core APIs implemented for each of these buses. Now, you will use the regmap API to do so. The regmap subsystem takes care of calling the relevant calls of the SPI or I2C subsystem.

The regmap subsystem mainly includes `struct regmap` (declared in `drivers/base/regmap/internal.h`), `struct regmap_bus` (declared in `include/linux/regmap.h`) and `struct regmap_config` (declared in `include/linux/regmap.h`). The `regmap` structure represents the mapping of the register operation of a slow i/o device, and the `regmap_bus` structure represents the register operation of a type of slow i/o device (for example, an SPI or an I2C device). The `regmap_bus` structure is bound to the `regmap` structure. The `regmap_config` structure is a per device and bus configuration structure used by the regmap subsystem to communicate with the device. It is defined by the driver code and contains all the information related to the registers of the device.

A device that can be accessed using an SPI or an I2C bus, such as the ADXL345 accelerometer, is a good candidate to use the regmap API. For this ADXL345 device, you can develop two simple drivers using the regmap API, one for I2C bus support (`adxl345-i2c.c`) and another for SPI bus support (`adxl345-spi.c`), writing specific code for each bus. This specific code configures the bus using a `regmap_config` structure and initializes struct `regmap` using the functions `devm_regmap_init_spi()` or `devm_regmap_init_i2c()`.

The `devm_regmap_init_spi()` function initializes a regmap data structure, taking as a first parameter an `spi_device` structure and as a second parameter a `regmap_config` structure (defined in our driver code) which has been configured to access to the SPI registers of the device.

```
struct regmap * devm_regmap_init_spi(struct spi_device *spi, const struct regmap_config);
```

The `devm_regmap_init_i2c()` function initializes a regmap data structure, taking as a first parameter an `i2c_client` structure and as a second parameter a `regmap_config` structure (defined in our driver code) which has been configured to access to the I2C registers of the device.

```
struct regmap * devm_regmap_init_i2c(struct i2c_client *i2c, const struct regmap_config);
```

The `devm_regmap_init_spi()` function, for example, will initialize struct `regmap` with the `regmap_spi` structure, which has been initialized by the regmap subsystem with specific functions that will call to the corresponding functions of the SPI subsystem:

```
static const struct regmap_bus regmap_spi = {
    .write = regmap_spi_write,
    .gather_write = regmap_spi_gather_write,
    .async_write = regmap_spi_async_write,
    .async_alloc = regmap_spi_async_alloc,
    .read = regmap_spi_read,
    .read_flag_mask = 0x80,
    .reg_format_endian_default = REGMAP_ENDIAN_BIG,
    .val_format_endian_default = REGMAP_ENDIAN_BIG,
};
```

The `regmap_spi_write()` function will call the `spi_write()` function, which in turn calls `spi_sync_transfer()`. You can see below the code of the `regmap_spi_write()` and `spi_write()` functions:

```
static int regmap_spi_write(void *context, const void *data, size_t count)
{
    struct device *dev = context;
    struct spi_device *spi = to_spi_device(dev);

    return spi_write(spi, data, count);
}

static inline int spi_write(struct spi_device *spi, const void *buf, size_t len)
{
    struct spi_transfer t = {
        .tx_buf = buf,
```

```

        .len = len,
    };

    return spi_sync_transfer(spi, &t, 1);
}

```

In the two previous regmap initialization routines, the regmap_config configuration is taken, then the regmap structure is allocated, and the configuration is copied to it. The read/write functions of the respective buses are also copied in the regmap structure.

See below the main lines of code of the proposed adxl345-i2c.c driver for the I2C register map configuration and initialization:

```

static const struct regmap_config adxl345_i2c_regmap_config = {
    .reg_bits = 8,
    .val_bits = 8,
};

static int adxl345_i2c_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    struct regmap *regmap;
    regmap = devm_regmap_init_i2c(client, &adxl345_i2c_regmap_config);
    return adxl345_core_probe(&client->dev, regmap, id ? id->name : NULL);
}

```

See below the main lines of code of the proposed adxl345-spi.c driver for the SPI register map configuration and initialization:

```

static const struct regmap_config adxl345_spi_regmap_config = {
    .reg_bits = 8,
    .val_bits = 8,
    /* Setting bits 7 and 6 enables multiple-byte read */
    .read_flag_mask = BIT(7) | BIT(6),
};

static int adxl345_spi_probe(struct spi_device *spi)
{
    const struct spi_device_id *id = spi_get_device_id(spi);
    struct regmap *regmap;

    regmap = devm_regmap_init_spi(spi, &adxl345_spi_regmap_config);

    return adxl345_core_probe(&spi->dev, regmap, id->name);
}

```

After you implement the regmap configuration and initialization using two specific SPI and I2C drivers, you should develop a common core driver (adxl345-accel-core.c) that can talk to the device using the following functions:

```

int regmap_write(struct regmap *map, unsigned int reg, unsigned int val);
int regmap_read(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);
```

As you have seen before, the `regmap_config` structure is the configuration for the register map of a device. Its most important fields are described below:

- **reg_bits**: This is the number of bits in the registers of the device, e.g., in case of 1 byte registers it will be set to the value 8.
- **val_bits**: This is the number of bits in the value that will be set in the device register.
- **writeable_reg**: This is an optional callback function written in driver code, which is called whenever a register is to be written. Whenever the driver calls the regmap subsystem to write to a register, this driver function is called; it will return "false" if this register is not writeable and the write operation will return an error to the driver.
- **wr_table**: If the driver does not provide the `writeable_reg` callback, then `wr_table` is checked by regmap before doing the write operation. If the register address lies in the range provided by the `wr_table`, then the write operation is performed. This is also optional, and the driver can omit its definition and can set it to NULL.
- **readable_reg**: This is an optional callback function written in driver code, which is called whenever a register is to be read. Whenever the driver calls the regmap subsystem to read a register, this driver function is called to ensure the register is readable. The driver function will return "false" if this register is not readable, and the read operation will return an error to the driver.
- **rd_table**: If a driver does not provide a `readable_reg` callback, then the `rd_table` is checked by regmap before doing the read operation. If the register address lies in the range provided by `rd_table`, then the read operation is performed. This is also optional, and the driver can omit its definition and can set it to NULL.
- **reg_read**: Optional callback that if filled will be used to perform all the reads from the registers. Should only be provided for devices whose read operation cannot be represented as a simple read operation on a bus such as SPI, I2C, etc. Most of the devices do not need this.
- **reg_write**: Same as above for writing.
- **volatile_reg**: This is a callback function called whenever a register is written or read through the cache. Whenever a driver reads or writes a register through the regmap cache, this function is called first, and if it returns "false" only then is the cache method used; else, the registers are written or read directly, since the register is volatile and caching is not to be used.

- **volatile_table:** If a driver does not provide a `volatile_reg` callback, then the `volatile_table` is checked by regmap to see if the register is volatile or not. If the register address lies in the range provided by the `volatile_table` then the cache operation is not used. This is also optional, and the driver can omit its definition and can set it to NULL.
- **max_register:** Whenever any read or write operation is to be performed, regmap checks whether the register address is less than `max_register` first, and only if it is, is the operation performed. The `max_register` is ignored if it is set to 0.
- **read_flag_mask:** Normally, in SPI or I2C, a write or read transaction will have the highest bit set in the top byte to differentiate write and read operations. This mask is set in the higher byte of the register value.
- **write_flag_mask:** This mask is also set in the higher byte of the register value. If both `read_flag_mask` and `write_flag_mask` are empty, the `regmap_bus` default masks are used.

The regmap infrastructure also provides APIs that are declared in `include/linux/regmap.h` and defined under `drivers/base/regmap/`. The following are the details of the `regmap_write` and the `regmap_read` APIs, taken from the very informative opensource article located at <https://opensourceforu.com/2017/01/regmap-reducing-redundancy-linux-code/>.

1. **regmap_write:** This function is used to write data to the device. It takes in the regmap structure returned during initialization and registers the address and the value to be set. The following are the steps performed by the `regmap_write` routine:
 - First, `regmap_write` takes the lock, which will be spinlock if `fast_io` in `regmap_config` was set; else, it will be mutex.
 - Next, if `max_register` is set in `regmap_config`, the function checks the register address to be written. If the register address is less than or equal to `max_register`, the write operation will be executed; otherwise, the regmap core will return an invalid I/O error.
 - After that, if the `writable_reg` callback is set in `regmap_config`, then that callback is called. If that callback returns "true", then further operations are done; if it returns "false", then an error -EIO is returned.
 - If `writable_reg` is not set, but `wr_table` is set, then there's a check on whether the register address lies in `no_ranges`, in which case an -EIO error is returned; else, it is checked whether it lies in the `yes_ranges`. If it is not present there, then an -EIO error is returned and the operation is terminated. If it lies in the `yes_ranges`, then further operations are performed. This step is only performed if `wr_table` is set; else, it is skipped.

- Whether caching is permitted is now checked. If it is permitted, then the register value is cached instead of writing directly to hardware, and the operation finishes at this step. If caching is not permitted, it goes to the next step.
 - After the above steps are taken, the hardware write routine (for example, `regmap_spi_write()`) is called to write the value in the hardware register. This function writes the `write_flag_mask` to the first byte of the value, and the value is written to the device.
 - After completing the write operation, the lock that was taken before writing is released, and the function returns.
2. **regmap_read:** This function is used to read data from the device. It takes in the `regmap` structure returned during initialization and registers the address and a pointer to the variable in which the data is to be read. The following are the steps performed by the `regmap_read` routine:
- First, the read function will take a lock before performing the read operation. This will be a spinlock if `fast_io` is set in `regmap_config`; else, `regmap` will use mutex.
 - Next, it will check whether the passed register address is less than `max_register`; if it is not, then `-EIO` is returned. This step is only done if `max_register` is set greater than zero.
 - Then, it will check if the `readable_reg` callback is set. If it is, then that callback is called, and if this callback returns "false", the read operation is terminated returning an `-EIO` error. If this callback returns "true", then further operations are performed. This step is only performed if `readable_reg` is set.
 - What is checked next is whether the register address lies in the `no_ranges` of the `rd_table` in `config`. If it does, then an `-EIO` error is returned. If it doesn't lie either in the `no_ranges` or in the `yes_ranges`, then too an `-EIO` error is returned. Only if it lies in the `yes_ranges` can further operations be performed. This step is only performed if the `rd_table` is set.
 - Now, if caching is permitted, then the register value is read from the cache and the function returns the value being read. If caching is set to `bypass`, then the next step is performed.
 - After the above steps have been taken, the hardware read operation (for example, `regmap_spi_read()`) is called to read the register value, and the value of the variable which was passed is updated with the value returned.
 - The lock that was taken before starting this operation is now released, and the function returns.

LAB 12.1: "SPI regmap IIO device" module

In this lab, you will develop a driver with similar functionality to the one developed in LAB 10.4, but this time, you will use the IIO subsystem instead of the Input subsystem to develop it. You will also access to the registers of the ADXL345 device by using the regmap API instead of the SPI specific core APIs.

As in the LAB 10.4 driver, this new driver will detect single tap motion on any of the three axes. The tap detection threshold is defined by the THRESH_TAP register (Address 0x1D). The SINGLE_TAP bit of the INT_SOURCE register (Address 0x30) is set when a single acceleration event greater than the value in the THRESH_TAP register (Address 0x1D) occurs in less time than is specified in the DUR register (Address 0x21). The single tap interrupt is triggered when the acceleration goes below the threshold, as long as DUR has not been exceeded (see page 28 of the ADXL345 data-sheet). The tap motion detection will be communicated to user space by sending an IIO event through the `iio_push_event()` function, which is called within the interrupt handler of the driver. You will set the value of the THRESH_TAP and DUR registers by writing from user space to the event sysfs attributes under `/sys/bus/iio/devices/iio:deviceX/events/` directory.

You will also create an IIO trigger buffer, which is used to store the values of the three axes (plus a timestamp value) captured by an IIO trigger (`iio-trig-hrtimer` or `iio-trig-sysfs` trigger) in each of the IIO buffer entries.

You will use the same HW and DT setup of the LAB 10.4.

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/module.h>
#include <linux/regmap.h>
#include <linux/spi/spi.h>
#include <linux/of_gpio.h>
#include <linux/iio/events.h>
#include <linux/iio/buffer.h>
#include <linux/iio/trigger_consumer.h>
#include <linux/iio/triggered_buffer.h>
```

2. Define the registers of the ADXL345 device:

```
/* ADXL345 Register Map */
#define DEVID                0x00    /* R   Device ID */
#define THRESH_TAP           0x1D    /* R/W Tap threshold */
#define DUR                  0x21    /* R/W Tap duration */
#define TAP_AXES             0x2A    /* R/W Axis control for tap/double tap */
#define ACT_TAP_STATUS       0x2B    /* R   Source of tap/double tap */
#define BW_RATE              0x2C    /* R/W Data rate and power mode control */
#define POWER_CTL            0x2D    /* R/W Power saving features control */
#define INT_ENABLE           0x2E    /* R/W Interrupt enable control */
```

```

#define INT_MAP                0x2F    /* R/W Interrupt mapping control */
#define INT_SOURCE              0x30    /* R   Source of interrupts */
#define DATA_FORMAT           0x31    /* R/W Data format control */
#define DATAX0                  0x32    /* R   X-Axis Data 0 */
#define DATAX1                  0x33    /* R   X-Axis Data 1 */
#define DATAY0                  0x34    /* R   Y-Axis Data 0 */
#define DATAY1                  0x35    /* R   Y-Axis Data 1 */
#define DATAZ0                  0x36    /* R   Z-Axis Data 0 */
#define DATAZ1                  0x37    /* R   Z-Axis Data 1 */
#define FIFO_CTL                0x38    /* R/W FIFO control */
#define FIFO_STATUS             0x39    /* R   FIFO status */

```

3. Create the rest of #define to perform operations in the ADXL345 registers and to pass some of them as arguments to several functions of the driver:

```

#define ADXL345_GPIO_NAME      "int"

/* DEVIDs */
#define ID_ADXL345              0xE5

/* INT_ENABLE/INT_MAP/INT_SOURCE Bits */
#define SINGLE_TAP              (1 << 6)
#define WATERMARK               (1 << 1)

/* TAP_AXES Bits */
#define TAP_X_EN                (1 << 2)
#define TAP_Y_EN                (1 << 1)
#define TAP_Z_EN                (1 << 0)

/* BW_RATE Bits */
#define LOW_POWER               (1 << 4)
#define RATE(x)                 ((x) & 0xF)

/* POWER_CTL Bits */
#define PCTL_MEASURE             (1 << 3)
#define PCTL_STANDBY            0X00

/* DATA_FORMAT Bits */
#define ADXL_FULL_RES            (1 << 3)

/* FIFO_CTL Bits */
#define FIFO_MODE(x)             (((x) & 0x3) << 6)
#define FIFO_BYPASS              0
#define FIFO_FIFO                1
#define FIFO_STREAM              2
#define SAMPLES(x)               ((x) & 0x1F)

/* FIFO_STATUS Bits */
#define ADXL_X_AXIS              0
#define ADXL_Y_AXIS              1
#define ADXL_Z_AXIS              2

/* Interrupt AXIS Enable */

```

```
#define ADXL_TAP_X_EN      (1 << 2)
#define ADXL_TAP_Y_EN      (1 << 1)
#define ADXL_TAP_Z_EN      (1 << 0)
```

4. Create a private `adxl345_data` structure:

```
struct adxl345_data {
    struct gpio_desc *gpio;
    struct regmap *regmap;
    struct iio_trigger *trig;
    struct device *dev;
    struct axis_triple saved;
    u8 data_range;
    u8 tap_threshold;
    u8 tap_duration;
    u8 tap_axis_control;
    u8 data_rate;
    u8 fifo_mode;
    u8 watermark;
    u8 low_power_mode;
    int irq;
    int ev_enable;
    u32 int_mask;
    s64 timestamp;
};
```

5. Create the `iio_chan_spec` and `iio_event_spec` structures to expose to user space the channel and event sysfs attributes. The `scan_index` variable defines the order in which the enabled channels are placed inside the IIO trigger buffer. The channels with a lower `scan_index` will be placed before the channels with a higher index. Each channel needs to have a unique `scan_index`.

```
/*
 * Each axis will have two event sysfs attributes
 * You will set THRESH_TAP register value associated to the specific axis
 * writing to the sysfs attribute with bitmask IIO_EV_INFO_VALUE
 * You will modify DUR register associated to the specific axis writing to the
 * sysfs attribute with bitmask IIO_EV_INFO_PERIOD
 * The THRESH_TAP and DUR registers are shared for all the axis so it
 * could have had more sense to use mask_shared_by_type instead mask_separate
 */
static const struct iio_event_spec adxl345_event = {
    .type = IIO_EV_TYPE_THRESH,
    .dir = IIO_EV_DIR_EITHER,
    .mask_separate = BIT(IIO_EV_INFO_VALUE) |
                    BIT(IIO_EV_INFO_PERIOD)
};

/*
 * Each axis will have is own channel sysfs attribute and there are two shared
 * sysfs attributes for the IIO_ACCEL type
 * You will get each axis value reading each channel sysfs attribute with
```

```

* bitmask IIO_CHAN_INFO_RAW
* There is a shared attribute to read the scale value with bitmask
* IIO_CHAN_INFO_SCALE
* There is a shared attribute to write the accel data rate with bitmask
* IIO_CHAN_INFO_SAMP_FREQ
*/
#define ADXL345_CHANNEL(reg, axis, idx) {
    .type = IIO_ACCEL,
    .modified = 1,
    .channel2 = IIO_MOD_##axis,
    .address = reg,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE) |
                                BIT(IIO_CHAN_INFO_SAMP_FREQ),
    .scan_index = idx,
    .scan_type = {
        .sign = 's',
        .realbits = 13,
        .storagebits = 16,
        .endianness = IIO_LE,
    },
    .event_spec = &adxl345_event,
    .num_event_specs = 1
}

static const struct iio_chan_spec adxl345_channels[] = {
    ADXL345_CHANNEL(DATA_X0, X, 0),
    ADXL345_CHANNEL(DATA_Y0, Y, 1),
    ADXL345_CHANNEL(DATA_Z0, Z, 2),
    IIO_CHAN_SOFT_TIMESTAMP(3),
};

```

6. Create the `iio_info` structure to declare the hooks that the IIO core will use for this device. There are four hooks available, corresponding to user space interactions through the channel and event sysfs attributes.

```

static const struct iio_info adxl345_info = {
    .driver_module      = THIS_MODULE,
    .read_raw           = adxl345_read_raw,
    .write_raw          = adxl345_write_raw,
    .read_event_value   = adxl345_read_event,
    .write_event_value  = adxl345_write_event,
};

```

See below a brief description of each of these callback functions:

- **adxl345_read_raw**: This function returns an axis value when a channel sysfs attribute with bitmask `IIO_CHAN_INFO_RAW` is read from user space. It also returns the accelerometer scale when the shared sysfs attribute with bitmask `IIO_CHAN_INFO_SCALE` is read from user space. You can see in the code below the `regmap_bulk_read()` function, which is used to access via SPI to the two registers of each axis.

```

static int adxl345_read_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int *val, int *val2, long mask)
{
    struct adxl345_data *data = iio_priv(indio_dev);
    __le16 regval;

    switch (mask) {
    case IIO_CHAN_INFO_RAW:
        regmap_bulk_read(data->regmap, chan->address, &regval, sizeof(regval));

        *val = sign_extend32(le16_to_cpu(regval), 12);

        return IIO_VAL_INT;

    case IIO_CHAN_INFO_SCALE:
        *val = 0;
        *val2 = adxl345_uscale;
        return IIO_VAL_INT_PLUS_MICRO;

    default:
        return -EINVAL;
    }
}

```

- **adxl345_write_raw:** This function sets the data rate and power mode control of the ADXL345 device (BW_RATE register) whenever user space writes to the shared sysfs attribute with bitmask IIO_CHAN_INFO_SAMP_FREQ. You can see in the following code snippet the regmap_write() function used to access via SPI to the register BW_RATE of the ADXL345 device.

```

static int adxl345_write_raw(struct iio_dev *indio_dev,
                             struct iio_chan_spec const *chan,
                             int val, int val2, long mask)
{
    struct adxl345_data *data = iio_priv(indio_dev);

    switch (mask) {
    case IIO_CHAN_INFO_SAMP_FREQ:
        data->data_rate = RATE(val);
        return regmap_write(data->regmap, BW_RATE, data->data_rate |
                             (data->low_power_mode ? LOW_POWER : 0));

    default :
        return -EINVAL;
    }
}

```

- **adxl345_read_event:** This function returns the value of the THRESH_TAP register or the DUR register whenever a sysfs attribute with bitmask IIO_EV_INFO_VALUE or IIO_EV_INFO_PERIOD is read from user space.

```

static int adxl345_read_event(struct iio_dev *indio_dev,
                             const struct iio_chan_spec *chan,
                             enum iio_event_type type,
                             enum iio_event_direction dir,
                             enum iio_event_info info,
                             int *val, int *val2)
{
    struct adxl345_data *data = iio_priv(indio_dev);

    switch (info) {
    case IIO_EV_INFO_VALUE:
        *val = data->tap_threshold;
        break;
    case IIO_EV_INFO_PERIOD:
        *val = data->tap_duration;
        break;
    default:
        return -EINVAL;
    }

    return IIO_VAL_INT;
}

```

- **adxl345_write_event**: This function sets the value of the THRESH_TAP register or the DUR register whenever a sysfs attribute with bitmask IIO_EV_INFO_VALUE or IIO_EV_INFO_PERIOD is written from user space.

```

static int adxl345_write_event(struct iio_dev *indio_dev,
                              const struct iio_chan_spec *chan,
                              enum iio_event_type type,
                              enum iio_event_direction dir,
                              enum iio_event_info info,
                              int val, int val2)
{
    struct adxl345_data *data = iio_priv(indio_dev);

    switch (info) {
    case IIO_EV_INFO_VALUE:
        data->tap_threshold = val;
        return regmap_write(data->regmap, THRESH_TAP,
                           data->tap_threshold);

    case IIO_EV_INFO_PERIOD:
        data->tap_duration = val;
        return regmap_write(data->regmap, DUR, data->tap_duration);
    default:
        return -EINVAL;
    }
}

```

7. See below in bold the main lines of code for the SPI register map configuration and initialization:


```

static const struct regmap_config adxl345_spi_regmap_config = {
    .reg_bits = 8,
    .val_bits = 8,
    /* Setting bits 7 and 6 enables multiple-byte read */
    .read_flag_mask = BIT(7) | BIT(6),
};

static int adxl345_spi_probe(struct spi_device *spi)
{
    struct regmap *regmap;

    /* get the id from the driver structure to use the name */
    const struct spi_device_id *id = spi_get_device_id(spi);

    regmap = devm_regmap_init_spi(spi, &adxl345_spi_regmap_config);

    return adxl345_core_probe(&spi->dev, regmap, id->name);
}

```

8. In the `adxl345_core_probe()` routine request a threaded interrupt. You will add a threaded interrupt to the driver to service the single tap interrupt. In a threaded interrupt, the `adxl345_event_handler` interrupt handler is executed inside a thread. It is allowed to block during the interrupt handler, which is often needed for SPI devices, as the interrupt handler needs to communicate with them. In the interrupt handler, you will communicate via SPI with the ADXL345 device by using the `regmap_read()` function. The `SINGLE_TAP` events will be sent to user space by using the `iio_push_event()` function.

```

/* Request threaded interrupt */
devm_request_threaded_irq(dev, data->irq, NULL, adxl345_event_handler,
                          IRQF_TRIGGER_HIGH | IRQF_ONESHOT, dev_name(dev), indio_dev);

/* Interrupt service routine */
static irqreturn_t adxl345_event_handler(int irq, void *handle)
{
    u32 tap_stat, int_stat;
    struct iio_dev *indio_dev = handle;
    struct adxl345_data *data = iio_priv(indio_dev);
    data->timestamp = iio_get_time_ns(indio_dev);

    if (data->tap_axis_control & (TAP_X_EN | TAP_Y_EN | TAP_Z_EN)) {
        regmap_read(data->regmap, ACT_TAP_STATUS, &tap_stat);
    }
    else
        tap_stat = 0;

    /*
     * Read the INT_SOURCE (0x30) register
     * The tap interrupt is cleared
     */
}

```

```

regmap_read(data->regmap, INT_SOURCE, &int_stat);

/*
 * if the SINGLE_TAP event has occurred the axl345_do_tap function
 * is called with the ACT_TAP_STATUS register as an argument
 */
if (int_stat & (SINGLE_TAP)) {
    dev_info(data->dev, "single tap interrupt has occurred\n");

    if (tap_stat & TAP_X_EN) {
        iio_push_event(indio_dev,
                        IIO_MOD_EVENT_CODE(IIO_ACCEL,
                                            0,
                                            IIO_MOD_X,
                                            IIO_EV_TYPE_THRESH,
                                            0),
                        data->timestamp);
    }
    if (tap_stat & TAP_Y_EN) {
        iio_push_event(indio_dev,
                        IIO_MOD_EVENT_CODE(IIO_ACCEL,
                                            0,
                                            IIO_MOD_Y,
                                            IIO_EV_TYPE_THRESH,
                                            0),
                        data->timestamp);
    }
    if (tap_stat & TAP_Z_EN) {
        iio_push_event(indio_dev,
                        IIO_MOD_EVENT_CODE(IIO_ACCEL,
                                            0,
                                            IIO_MOD_Z,
                                            IIO_EV_TYPE_THRESH,
                                            0),
                        data->timestamp);
    }
}

return IRQ_HANDLED;
}

```

9. In the `adxl345_core_probe()` routine, allocate an IIO trigger buffer by using the `devm_iio_triggered_buffer_setup()` function. This function combines some common tasks which will normally be performed when setting up a triggered buffer. It allocates the buffer and sets the "pollfunc top half" and the "pollfunc bottom half" handlers. The `adxl345_trigger_handler` (pollfunc bottom half) handler runs in the context of a kernel thread, and all the processing takes place here. It reads the data of the three axes from the ADXL345 device and stores the values in the internal buffer (together with the timestamp obtained in the top half) by using the `iio_push_to_buffers_with_timestamp()` function. The pollfunc top half should do as little processing as possible because it runs in interrupt

context. The most common operation is the recording of the current timestamp, and for this reason, you can use the `iio_pollfunc_store_time()` function. Before calling `devm_iio_triggered_buffer_setup()`, the `indio_dev` structure should already be completely initialized but not registered yet. In practice, this means that this function should be called right before `devm_iio_device_register()`.

```
int adxl345_core_probe(struct device *dev, struct regmap *regmap, const char *name)
{
    struct iio_dev *indio_dev;
    struct adxl345_data *data;

    [...]

    /* iio_pollfunc_store_time do pf->timestamp = iio_get_time_ns(); */
    devm_iio_triggered_buffer_setup(dev, indio_dev,
                                   &iio_pollfunc_store_time,
                                   adxl345_trigger_handler, NULL);

    devm_iio_device_register(dev, indio_dev);

    return 0;
}

static irqreturn_t adxl345_trigger_handler(int irq, void *p)
{
    struct iio_poll_func *pf = p;
    struct iio_dev *indio_dev = pf->indio_dev;
    struct adxl345_data *data = iio_priv(indio_dev);

    /* 6 bytes axis + 2 bytes padding + 8 bytes timestamp */
    s16 buf[8];
    int i, ret, j = 0, base = DATA0;
    s16 sample;

    /* Read the channels that have been enabled from user space */
    for_each_set_bit(i, indio_dev->active_scan_mask, indio_dev->masklength) {
        ret = regmap_bulk_read(data->regmap, base + i * sizeof(sample),
                               &sample, sizeof(sample));

        if (ret < 0)
            goto done;
        buf[j++] = sample;
    }

    iio_push_to_buffers_with_timestamp(indio_dev, buf, pf->timestamp);

done:
    iio_trigger_notify_done(indio_dev->trig);

    return IRQ_HANDLED;
}
```

10. Declare a list of devices supported by the driver:

```
static const struct of_device_id adxl345_dt_ids[] = {
    { .compatible = "arrow,adxl345", },
    { }
};
MODULE_DEVICE_TABLE(of, adxl345_dt_ids);
```

11. Define an array of `spi_device_id` structures:

```
static const struct spi_device_id adxl345_id[] = {
    { .name = "adxl345", },
    { }
};
MODULE_DEVICE_TABLE(spi, adxl345_id);
```

12. Add an `spi_driver` structure that will be registered to the SPI bus:

```
static struct spi_driver adxl345_driver = {
    .driver = {
        .name = "adxl345",
        .owner = THIS_MODULE,
        .of_match_table = adxl345_dt_ids,
    },
    .probe =      adxl345_spi_probe,
    .remove =     adxl345_spi_remove,
    .id_table =   adxl345_id,
};
```

13. Register your driver with the SPI bus:

```
module_spi_driver(adxl345_driver);
```

14. Create a new `adxl345_rpi3_iio.c` file in the `linux_5.4_rpi3_drivers` folder, and add `adxl345_rpi3_iio.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers$ make
~/linux_5.4_rpi3_drivers$ make deploy
```

Listing 12-1: adxl345_rpi3_iio.c

```

#include <linux/module.h>
#include <linux/regmap.h>
#include <linux/spi/spi.h>
#include <linux/of_gpio.h>
#include <linux/iio/events.h>
#include <linux/iio/buffer.h>
#include <linux/iio/trigger.h>
#include <linux/iio/trigger_consumer.h>
#include <linux/iio/triggered_buffer.h>

/* ADXL345 Register Map */
#define DEVID                0x00    /* R   Device ID */
#define THRESH_TAP           0x1D    /* R/W Tap threshold */
#define DUR                  0x21    /* R/W Tap duration */
#define TAP_AXES             0x2A    /* R/W Axis control for tap/double tap */
#define ACT_TAP_STATUS       0x2B    /* R   Source of tap/double tap */
#define BW_RATE              0x2C    /* R/W Data rate and power mode control */
#define POWER_CTL            0x2D    /* R/W Power saving features control */
#define INT_ENABLE           0x2E    /* R/W Interrupt enable control */
#define INT_MAP              0x2F    /* R/W Interrupt mapping control */
#define INT_SOURCE           0x30    /* R   Source of interrupts */
#define DATA_FORMAT         0x31    /* R/W Data format control */
#define DATA0               0x32    /* R   X-Axis Data 0 */
#define DATA1               0x33    /* R   X-Axis Data 1 */
#define DATA0               0x34    /* R   Y-Axis Data 0 */
#define DATA1               0x35    /* R   Y-Axis Data 1 */
#define DATA0               0x36    /* R   Z-Axis Data 0 */
#define DATA1               0x37    /* R   Z-Axis Data 1 */
#define FIFO_CTL             0x38    /* R/W FIFO control */
#define FIFO_STATUS          0x39    /* R   FIFO status */

enum adxl345_accel_axis {
    AXIS_X,
    AXIS_Y,
    AXIS_Z,
    AXIS_MAX,
};

#define ADXL345_GPIO_NAME    "int"

/* DEVIDs */
#define ID_ADXL345           0xE5

/* INT_ENABLE/INT_MAP/INT_SOURCE Bits */
#define SINGLE_TAP           (1 << 6)
#define WATERMARK            (1 << 1)

/* TAP_AXES Bits */
#define TAP_X_EN             (1 << 2)
#define TAP_Y_EN             (1 << 1)
#define TAP_Z_EN             (1 << 0)

```

```

/* BW_RATE Bits */
#define LOW_POWER          (1 << 4)
#define RATE(x)            ((x) & 0xF)

/* POWER_CTL Bits */
#define PCTL_MEASURE        (1 << 3)
#define PCTL_STANDBY        0X00

/* DATA_FORMAT Bits */
#define ADXL_FULL_RES        (1 << 3)

/* FIFO_CTL Bits */
#define FIFO_MODE(x)        (((x) & 0x3) << 6)
#define FIFO_BYPASS          0
#define FIFO_FIFO            1
#define FIFO_STREAM          2
#define SAMPLES(x)          ((x) & 0x1F)

/* FIFO_STATUS Bits */
#define ADXL_X_AXIS          0
#define ADXL_Y_AXIS          1
#define ADXL_Z_AXIS          2

/* Interrupt AXIS Enable */
#define ADXL_TAP_X_EN        (1 << 2)
#define ADXL_TAP_Y_EN        (1 << 1)
#define ADXL_TAP_Z_EN        (1 << 0)

static const int adxl345_uscale = 38300;

struct axis_triple {
    int x;
    int y;
    int z;
};

struct adxl345_data {
    struct gpio_desc *gpio;
    struct regmap *regmap;
    struct iio_trigger *trig;
    struct device *dev;
    struct axis_triple saved;
    u8 data_range;
    u8 tap_threshold;
    u8 tap_duration;
    u8 tap_axis_control;
    u8 data_rate;
    u8 fifo_mode;
    u8 watermark;
    u8 low_power_mode;
    int irq;
    int ev_enable;
    u32 int_mask;

```

```

    s64 timestamp;
};

/* Set the events */
static const struct iio_event_spec adxl345_event = {
    .type = IIO_EV_TYPE_THRESH,
    .dir = IIO_EV_DIR_EITHER,
    .mask_separate = BIT(IIO_EV_INFO_VALUE) |
                     BIT(IIO_EV_INFO_PERIOD)
};

#define ADXL345_CHANNEL(reg, axis, idx) { \
    .type = IIO_ACCEL, \
    .modified = 1, \
    .channel2 = IIO_MOD_##axis, \
    .address = reg, \
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW), \
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE) | \
                                BIT(IIO_CHAN_INFO_SAMP_FREQ), \
    .scan_index = idx, \
    .scan_type = { \
        .sign = 's', \
        .realbits = 13, \
        .storagebits = 16, \
        .endianness = IIO_LE, \
    }, \
    .event_spec = &adxl345_event, \
    .num_event_specs = 1 \
}

static const struct iio_chan_spec adxl345_channels[] = {
    ADXL345_CHANNEL(DATA_X0, X, 0),
    ADXL345_CHANNEL(DATA_Y0, Y, 1),
    ADXL345_CHANNEL(DATA_Z0, Z, 2),
    IIO_CHAN_SOFT_TIMESTAMP(3),
};

static int adxl345_read_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int *val, int *val2, long mask)
{
    struct adxl345_data *data = iio_priv(indio_dev);
    __le16 regval;
    int ret;

    switch (mask) {
    case IIO_CHAN_INFO_RAW: /* Add an entry in the sysfs */

        /*
         * Data is stored in adjacent registers:
         * ADXL345_REG_DATA(X0/Y0/Z0) contain the least significant byte
         * and ADXL345_REG_DATA(X0/Y0/Z0) + 1 the most significant byte.
         * We are reading 2 bytes and storing in a __le16
         */

```

```

    ret = regmap_bulk_read(data->regmap, chan->address, &regval, sizeof(regval));
    if (ret < 0)
        return ret;

    *val = sign_extend32(le16_to_cpu(regval), 12);

    return IIO_VAL_INT;

case IIO_CHAN_INFO_SCALE: /* Add an entry in the sysfs */
    *val = 0;
    *val2 = adxl345_uscale;
    return IIO_VAL_INT_PLUS_MICRO;

default:
    return -EINVAL;
}
}

static int adxl345_write_raw(struct iio_dev *indio_dev,
                           struct iio_chan_spec const *chan,
                           int val, int val2, long mask)
{
    struct adxl345_data *data = iio_priv(indio_dev);

    switch (mask) {
case IIO_CHAN_INFO_SAMP_FREQ:
    data->data_rate = RATE(val);
    return regmap_write(data->regmap, BW_RATE,
                       data->data_rate | (data->low_power_mode ? LOW_POWER : 0));
default:
    return -EINVAL;
    }
}

static int adxl345_read_event(struct iio_dev *indio_dev,
                             const struct iio_chan_spec *chan,
                             enum iio_event_type type,
                             enum iio_event_direction dir,
                             enum iio_event_info info,
                             int *val, int *val2)
{
    struct adxl345_data *data = iio_priv(indio_dev);

    switch (info) {
case IIO_EV_INFO_VALUE:
    *val = data->tap_threshold;
    break;
case IIO_EV_INFO_PERIOD:
    *val = data->tap_duration;
    break;
default:
    return -EINVAL;
    }
}

```



```

    return IIO_VAL_INT;
}

static int adxl345_write_event(struct iio_dev *indio_dev,
                              const struct iio_chan_spec *chan,
                              enum iio_event_type type,
                              enum iio_event_direction dir,
                              enum iio_event_info info,
                              int val, int val2)
{
    struct adxl345_data *data = iio_priv(indio_dev);

    switch (info) {
    case IIO_EV_INFO_VALUE:
        data->tap_threshold = val;
        return regmap_write(data->regmap, THRESH_TAP, data->tap_threshold);

    case IIO_EV_INFO_PERIOD:
        data->tap_duration = val;
        return regmap_write(data->regmap, DUR, data->tap_duration);

    default:
        return -EINVAL;
    }
}

static const struct regmap_config adxl345_spi_regmap_config = {
    .reg_bits = 8,
    .val_bits = 8,
    /* Setting bits 7 and 6 enables multiple-byte read */
    .read_flag_mask = BIT(7) | BIT(6),
};

static const struct iio_info adxl345_info = {
    .driver_module      = THIS_MODULE,
    .read_raw           = adxl345_read_raw,
    .write_raw          = adxl345_write_raw,
    .read_event_value   = adxl345_read_event,
    .write_event_value  = adxl345_write_event,
};

/* Available channels, enabled from user space or using active_scan_mask */
static const unsigned long adxl345_accel_scan_masks[] = {
    BIT(Axis_X) | BIT(Axis_Y) | BIT(Axis_Z),
    0};

/* Interrupt service routine */
static irqreturn_t adxl345_event_handler(int irq, void *handle)
{
    u32 tap_stat, int_stat;
    int ret;
    struct iio_dev *indio_dev = handle;
    struct adxl345_data *data = iio_priv(indio_dev);

    data->timestamp = iio_get_time_ns(indio_dev);

```

```

/*
 * ACT_TAP_STATUS should be read before clearing the interrupt
 * Avoid reading ACT_TAP_STATUS in case TAP detection is disabled
 * Read the ACT_TAP_STATUS if any of the axis has been enabled
 */
if (data->tap_axis_control & (TAP_X_EN | TAP_Y_EN | TAP_Z_EN)) {
    ret = regmap_read(data->regmap, ACT_TAP_STATUS, &tap_stat);
    if (ret) {
        dev_err(data->dev, "error reading ACT_TAP_STATUS register\n");
        return ret;
    }
} else
    tap_stat = 0;

/*
 * Read the INT_SOURCE (0x30) register
 * to clear the tap interrupt
 */
ret = regmap_read(data->regmap, INT_SOURCE, &int_stat);
if (ret) {
    dev_err(data->dev, "error reading INT_SOURCE register\n");
    return ret;
}

/*
 * If the SINGLE_TAP event has occurred, the axl345_do_tap function
 * is called with the ACT_TAP_STATUS register as an argument
 */
if (int_stat & (SINGLE_TAP)) {
    dev_info(data->dev, "single tap interrupt has occurred\n");

    if (tap_stat & TAP_X_EN) {
        iio_push_event(indio_dev,
                       IIO_MOD_EVENT_CODE(IIO_ACCEL,
                                           0,
                                           IIO_MOD_X,
                                           IIO_EV_TYPE_THRESH,
                                           0),
                       data->timestamp);
    }
    if (tap_stat & TAP_Y_EN) {
        iio_push_event(indio_dev,
                       IIO_MOD_EVENT_CODE(IIO_ACCEL,
                                           0,
                                           IIO_MOD_Y,
                                           IIO_EV_TYPE_THRESH,
                                           0),
                       data->timestamp);
    }
    if (tap_stat & TAP_Z_EN) {
        iio_push_event(indio_dev,
                       IIO_MOD_EVENT_CODE(IIO_ACCEL,

```

```

                                0,
                                IIO_MOD_Z,
                                IIO_EV_TYPE_THRESH,
                                0),
                                data->timestamp);
        }
    }

    return IRQ_HANDLED;
}

static irqreturn_t adxl345_trigger_handler(int irq, void *p)
{
    struct iio_poll_func *pf = p;
    struct iio_dev *indio_dev = pf->indio_dev;
    struct adxl345_data *data = iio_priv(indio_dev);
    s16 buf[8] /* 16 bytes */
    int i, ret, j = 0, base = DATA0;
    s16 sample;

    /* Read the channels that have been enabled from user space */
    for_each_set_bit(i, indio_dev->active_scan_mask, indio_dev->masklength) {
        ret = regmap_bulk_read(data->regmap, base + i * sizeof(sample),
                                &sample, sizeof(sample));

        if (ret < 0)
            goto done;
        buf[j++] = sample;
    }

    /* Each buffer entry line is 6 bytes + 2 bytes pad + 8 bytes timestamp */
    iio_push_to_buffers_with_timestamp(indio_dev, buf, pf->timestamp);

done:
    iio_trigger_notify_done(indio_dev->trig);

    return IRQ_HANDLED;
}

int adxl345_core_probe(struct device *dev, struct regmap *regmap, const char *name)
{
    struct iio_dev *indio_dev;
    struct adxl345_data *data;
    u32 regval;
    int ret;

    ret = regmap_read(regmap, DEVID, &regval);
    if (ret < 0) {
        dev_err(dev, "Error reading device ID: %d\n", ret);
        return ret;
    }
    if (regval != ID_ADXL345) {
        dev_err(dev, "Invalid device ID: %x, expected %x\n",
                regval, ID_ADXL345);
        return -ENODEV;
    }
}

```

```

}

indio_dev = devm_iio_device_alloc(dev, sizeof(*data));
if (!indio_dev)
    return -ENOMEM;

/* Link private data with indio_dev */
data = iio_priv(indio_dev);
data->dev = dev;

/* Link spi device with indio_dev */
dev_set_drvdata(dev, indio_dev);

data->gpio = devm_gpiod_get_index(dev, ADXL345_GPIO_NAME, 0, GPIOD_IN);
if (IS_ERR(data->gpio)) {
    dev_err(dev, "gpio get index failed\n");
    return PTR_ERR(data->gpio);
}

data->irq = gpiod_to_irq(data->gpio);
if (data->irq < 0)
    return data->irq;
dev_info(dev, "The IRQ number is: %d\n", data->irq);

/* Initialize your private device structure */
data->regmap = regmap;
data->data_range = ADXL_FULL_RES;
data->tap_threshold = 50;
data->tap_duration = 3;
data->tap_axis_control = ADXL_TAP_Z_EN;
data->data_rate = 8;
data->fifo_mode = FIFO_BYPASS;
data->watermark = 32;
data->low_power_mode = 0;

indio_dev->dev.parent = dev;
indio_dev->name = name;
indio_dev->info = &adxl345_info;
indio_dev->modes = INDIO_DIRECT_MODE;
indio_dev->available_scan_masks = adxl345_accel_scan_masks;
indio_dev->channels = adxl345_channels;
indio_dev->num_channels = ARRAY_SIZE(adxl345_channels);

/* Initialize the ADXL345 registers */
/* 13-bit full resolution right justified */
ret = regmap_write(data->regmap, DATA_FORMAT, data->data_range);
if (ret < 0)
    goto error_standby;

/* Set the tap threshold and duration */
ret = regmap_write(data->regmap, THRESH_TAP, data->tap_threshold);
if (ret < 0)
    goto error_standby;

```

```

ret = regmap_write(data->regmap, DUR, data->tap_duration);
if (ret < 0)
    goto error_standby;

/* Set the axis where the tap will be detected */
ret = regmap_write(data->regmap, TAP_AXES, data->tap_axis_control);
if (ret < 0)
    goto error_standby;

/*
 * Set the data rate and the axis reading power
 * mode (higher noise less power).
 * In the initial settings is NO low power
 */
ret = regmap_write(data->regmap, BW_RATE,
                   RATE(data->data_rate) | (data->low_power_mode ? LOW_POWER : 0));
if (ret < 0)
    goto error_standby;

/* Set the FIFO mode, no FIFO by default */
ret = regmap_write(data->regmap, FIFO_CTL,
                   FIFO_MODE(data->fifo_mode) | SAMPLES(data->watermark));
if (ret < 0)
    goto error_standby;

/* Map all INTs to INT1 pin */
ret = regmap_write(data->regmap, INT_MAP, 0);
if (ret < 0)
    goto error_standby;

/* Enables interrupts */
if (data->tap_axis_control & (TAP_X_EN | TAP_Y_EN | TAP_Z_EN))
    data->int_mask |= SINGLE_TAP;

ret = regmap_write(data->regmap, INT_ENABLE, data->int_mask);
if (ret < 0)
    goto error_standby;

/* Enable measurement mode */
ret = regmap_write(data->regmap, POWER_CTL, PCTL_MEASURE);
if (ret < 0)
    goto error_standby;

/* Request threaded interrupt */
ret = devm_request_threaded_irq(dev, data->irq, NULL, adxl345_event_handler,
                                IRQF_TRIGGER_HIGH | IRQF_ONESHOT,
                                dev_name(dev), indio_dev);

if (ret) {
    dev_err(dev, "failed to request interrupt %d (%d)", data->irq, ret);
    goto error_standby;
}

dev_info(dev, "using interrupt %d", data->irq);

```

```

ret = devm_iio_triggered_buffer_setup(dev, indio_dev, &iio_pollfunc_store_time,
                                     adxl345_trigger_handler, NULL);
if (ret) {
    dev_err(dev, "unable to setup triggered buffer\n");
    goto error_standby;
}

ret = devm_iio_device_register(dev, indio_dev);
if (ret) {
    dev_err(dev, "iio_device_register failed: %d\n", ret);
    goto error_standby;
}

return 0;

error_standby:
dev_info(dev, "set standby mode due to an error\n");
regmap_write(data->regmap, POWER_CTL, PCTL_STANDBY);
return ret;
}

int adxl345_core_remove(struct device *dev)
{
    struct iio_dev *indio_dev = dev_get_drvdata(dev);
    struct adxl345_data *data = iio_priv(indio_dev);
    dev_info(data->dev, "my_remove() function is called.\n");
    return regmap_write(data->regmap, POWER_CTL, PCTL_STANDBY);
}

static int adxl345_spi_probe(struct spi_device *spi)
{
    struct regmap *regmap;

    /* get the id from the driver structure to use the name */
    const struct spi_device_id *id = spi_get_device_id(spi);

    regmap = devm_regmap_init_spi(spi, &adxl345_spi_regmap_config);
    if (IS_ERR(regmap)) {
        dev_err(&spi->dev, "Error initializing spi regmap: %ld\n",
                PTR_ERR(regmap));
        return PTR_ERR(regmap);
    }

    return adxl345_core_probe(&spi->dev, regmap, id->name);
}

static int adxl345_spi_remove(struct spi_device *spi)
{
    return adxl345_core_remove(&spi->dev);
}

static const struct spi_device_id adxl345_id[] = {
    { .name = "adxl345", },
    { }
}

```

```
};
MODULE_DEVICE_TABLE(spi, adxl345_id);

static const struct of_device_id adxl345_dt_ids[] = {
    { .compatible = "arrow,adxl345" },
    { },
};
MODULE_DEVICE_TABLE(of, adxl345_dt_ids);

static struct spi_driver adxl345_driver = {
    .driver = {
        .name      = "adxl345",
        .owner      = THIS_MODULE,
        .of_match_table = adxl345_dt_ids,
    },
    .probe          = adxl345_spi_probe,
    .remove         = adxl345_spi_remove,
    .id_table       = adxl345_id,
};

module_spi_driver(adxl345_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("ADXL345 Three-Axis Accelerometer Regmap SPI Bus Driver");
```

adxl345_rpi3_iio.ko demonstration

In the Linux host build the IIO tools. Open the Makefile under `~/linux_rpi3/linux/tools/iio/` folder, and add the following lines in bold:

```
~/linux_rpi3/linux/tools/iio$ gedit Makefile
# SPDX-License-Identifier: GPL-2.0
include ../scripts/Makefile.include
CC = arm-linux-gnueabi-gcc
LD = arm-linux-gnueabi-ld

~/linux_rpi3/linux/tools/iio$ ls
Build          iio_generic_buffer.c  iio_utils.h  Makefile
iio_event_monitor.c  iio_utils.c          lsiiio.c

~/linux_rpi3/linux/tools/iio$ make
```

Send `iio_generic_buffer` and `iio_event_monitor` applications to the Raspberry Pi:

```
~/linux_rpi3/linux/tools/iio$ scp iio_generic_buffer root@10.0.0.10:/home/pi
~/linux_rpi3/linux/tools/iio$ scp iio_event_monitor root@10.0.0.10:/home/pi
```

Load the module:

```
root@raspberrypi:/home/pi# insmod adxl345_rpi3_iio.ko
adxl345 spi0.0: The IRQ number is: 166
adxl345 spi0.0: using interrupt 166
```

See the entries under the `adxl345` IIO device:

```
root@raspberrypi:/home/pi# cd /sys/bus/iio/devices/iio:device0
root@raspberrypi:/sys/bus/iio/devices/iio:device0# ls
buffer          in_accel_x_raw  scan_elements
current_timestamp_clock  in_accel_y_raw  subsystem
dev             in_accel_z_raw  trigger
events          name            uevent
in_accel_sampling_frequency  of_node
in_accel_scale  power
```

Read the name and the scale of the IIO device:

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat name
adxl345
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_scale
0.038300
```

Move the `adxl345` board and read the `z` axis:

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_z_raw
-6
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cat in_accel_z_raw
240
```

Move the `adxl345` board until you generate some interrupts:


```
root@raspberrypi:/sys/bus/iio/devices/iio:device0#
adxl345 spi0.0: single tap interrupt has occurred
adxl345 spi0.0: single tap interrupt has occurred
adxl345 spi0.0: single tap interrupt has occurred
```

Read the values of the adxl345 axes using a sysfs trigger interface.

See the `iio_sysfs_trigger` folder:

```
root@raspberrypi:/sys/bus/iio/devices/iio:device0# cd /sys/bus/iio/devices/
root@raspberrypi:/sys/bus/iio/devices# ls
iio:device0 iio_sysfs_trigger
```

Create a sysfs trigger:

```
root@raspberrypi:/sys/bus/iio/devices# echo 1 > iio_sysfs_trigger/add_trigger
```

See the created `trigger0` folder:

```
root@raspberrypi:/sys/bus/iio/devices# ls
iio:device0 iio_sysfs_trigger trigger0
```

Attach the trigger to the iio device:

```
root@raspberrypi:/sys/bus/iio/devices# cat trigger0/name > iio:device0/trigger/current_trigger
```

Enable the scan elements:

```
root@raspberrypi:/sys/bus/iio/devices# echo 1 > iio:device0/scan_elements/in_accel_x_en
root@raspberrypi:/sys/bus/iio/devices# echo 1 > iio:device0/scan_elements/in_accel_y_en
root@raspberrypi:/sys/bus/iio/devices# echo 1 > iio:device0/scan_elements/in_accel_z_en
root@raspberrypi:/sys/bus/iio/devices# echo 1 > iio:device0/scan_elements/in_timestamp_en
```

Set the number of sample sets that may be held by the buffer:

```
root@raspberrypi:/sys/bus/iio/devices# echo 100 > iio:device0/buffer/length
```

Enable the buffer:

```
root@raspberrypi:/sys/bus/iio/devices# echo 1 > iio:device0/buffer/enable
```

Acquire two sample sets:

```
root@raspberrypi:/sys/bus/iio/devices# echo 1 > trigger0/trigger_now
root@raspberrypi:/sys/bus/iio/devices# echo 1 > trigger0/trigger_now
```

Display the acquired values: `tree axis + timestamp`:

```
root@raspberrypi:/sys/bus/iio/devices# hexdump -v -e '16/1 "%02x " "\n" < /dev/iio\:device0
ea ff ee 00 26 00 c0 b4 e0 61 5c 63 40 fd 73 16
ee ff fb 00 28 00 61 a6 00 a5 e2 c9 42 fd 73 16
```

Exit with `^C`:

```
root@raspberrypi:/sys/bus/iio/devices#
```

Disable the buffer:

```
root@raspberrypi:/sys/bus/iio/devices# echo 0 > iio:device0/buffer/enable
```

Detach the trigger:

```
root@raspberrypi:/sys/bus/iio/devices# echo "" > iio:device0/trigger/current_trigger
```

Remove the module:

```
root@raspberrypi:/sys/bus/iio/devices# cd /home/pi
root@raspberrypi:/home/pi# rmmod adxl345_rpi3_iio.ko
adxl345 spi0.0: my_remove() function is called.
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Use the `iio_generic_buffer` application to set the buffer length, the number of acquisitions and to enable the scan elements.

Load the module:

```
root@raspberrypi:/home/pi# insmod adxl345_rpi3_iio.ko
```

Create the sysfs trigger:

```
root@raspberrypi:/home/pi# echo 1 > /sys/bus/iio/devices/iio_sysfs_trigger/add_trigger
```

Attach the trigger to the device:

```
root@raspberrypi:/home/pi# echo sysfsstrig1 > /sys/bus/iio/devices/iio:device0/trigger/current_trigger
```

Launch the `iio_generic_buffer` application:

```
root@raspberrypi:/home/pi# ./iio_generic_buffer --device-num 0 -T 0 -a -l 10 -c 5 &
[1] 905
iio device number being used is 0
iio trigger number being used is 0
Enabling all channels
Enabling: in_accel_y_en
Enabling: in_accel_x_en
Enabling: in_timestamp_en
Enabling: in_accel_z_en
/sys/bus/iio/devices/iio:device0 sysfsstrig1
root@raspberrypi:/home/pi#
```

Execute the trigger 5 times:

```
root@raspberrypi:/home/pi# echo 1 > /sys/bus/iio/devices/trigger0/trigger_now
-1.953300 8.694100 1.149000 1619799734646166376
root@raspberrypi:/home/pi# echo 1 > /sys/bus/iio/devices/trigger0/trigger_now
0.268100 -0.459600 9.498400 1619799739996161374
root@raspberrypi:/home/pi# echo 1 > /sys/bus/iio/devices/trigger0/trigger_now
0.114900 9.077100 1.723500 1619799745306112414
root@raspberrypi:/home/pi# echo 1 > /sys/bus/iio/devices/trigger0/trigger_now
2.527800 -7.621700 6.664200 1619799750036123245
```

```
root@raspberrypi:/home/pi# echo 1 > /sys/bus/iio/devices/trigger0/trigger_now
2.336300 -6.894000 7.277000 1619799752036095796
```

```
root@raspberrypi:/home/pi# Disabling: in_accel_y_en
```

```
Disabling: in_accel_x_en
```

```
Disabling: in_timestamp_en
```

```
Disabling: in_accel_z_en
```

Exit with ^C:

```
root@raspberrypi:/home/pi#
```

Remove the module:

```
root@raspberrypi:/sys/bus/iio/devices# cd /home/pi
```

```
root@raspberrypi:/home/pi# rmmod adxl345_rpi3_iio.ko
```

Reboot the Raspberry Pi:

```
root@raspberrypi:/home/pi# reboot
```

Capture data using the hrtimer trigger.

Load the module:

```
root@raspberrypi:/home/pi# insmod adxl345_rpi3_iio.ko
```

Create the config folder:

```
root@raspberrypi:/home/pi# mkdir /config
```

Mount the configfs file system:

```
root@raspberrypi:/home/pi# mount -t configfs none /config
```

Create the hrtimer trigger:

```
root@raspberrypi:/home/pi# mkdir /config/iio/triggers/hrtimer/trigger0
```

Set the sampling frequency:

```
root@raspberrypi:/home/pi# echo 50 > /sys/bus/iio/devices/trigger0/sampling_frequency
```

Attach the trigger to the device:

```
root@raspberrypi:/home/pi# echo trigger0 > /sys/bus/iio/devices/iio:device0/trigger/current_trigger
```

Use the `iio_generic_buffer` application to set the buffer length and number of conversions; the application enables the scan elements, does the acquisitions and shows them; after that, disables the scan elements:

```
root@raspberrypi:/home/pi# ./iio_generic_buffer --device-num 0 -T 0 -a -l 10 -c 10 &
[1] 899
```

```
iio device number being used is 0
```

```
iio trigger number being used is 0
```

```
Enabling all channels
```

```
Enabling: in_accel_y_en
```

```

Enabling: in_accel_x_en
Enabling: in_timestamp_en
Enabling: in_accel_z_en
/sys/bus/iio/devices/iio:device0 trigger0
-0.957500 8.579200 1.608600 1619800613157990969
-0.957500 8.579200 1.608600 1619800613177988625
root@raspberrypi:/home/pi# -0.995800 8.655800 1.723500 1619800613197988834
-0.957500 8.694100 1.685200 1619800613217988938
-0.957500 8.694100 1.685200 1619800613237987844
-0.995800 8.847300 1.570300 1619800613257987896
-0.995800 8.847300 1.570300 1619800613277987844
-1.034100 8.847300 1.532000 1619800613297987740
-1.034100 8.847300 1.532000 1619800613317987740
-0.957500 8.962200 1.570300 1619800613337987792
Disabling: in_accel_y_en
Disabling: in_accel_x_en
Disabling: in_timestamp_en
Disabling: in_accel_z_en

```

Exit with ^C:

```
root@raspberrypi:/home/pi#
```

Execute the `iio_event_monitor` application, and move the accel board until you see several IIO events:

```

root@raspberrypi:/home/pi# ./iio_event_monitor /dev/iio\:device0
adxl345 spi0.0: single tap interrupt has occurred
Event: time: 1619800721034526397, type: accel(z), channel: 0, evtype: thresh, di
rection: either
adxl345 spi0.0: single tap interrupt has occurred
Event: time: 1619800722083346136, type: accel(z), channel: 0, evtype: thresh, di
rection: either
adxl345 spi0.0: single tap interrupt has occurred
Event: time: 1619800750157574979, type: accel(z), channel: 0, evtype: thresh, di
rection: either

```

Exit with ^C:

```
root@raspberrypi:/home/pi#
```

Remove the module:

```

root@raspberrypi:/home/pi# rmmod adxl345_rpi3_iio.ko
adxl345 spi0.0: my_remove() function is called.

```

13

USB Device Drivers

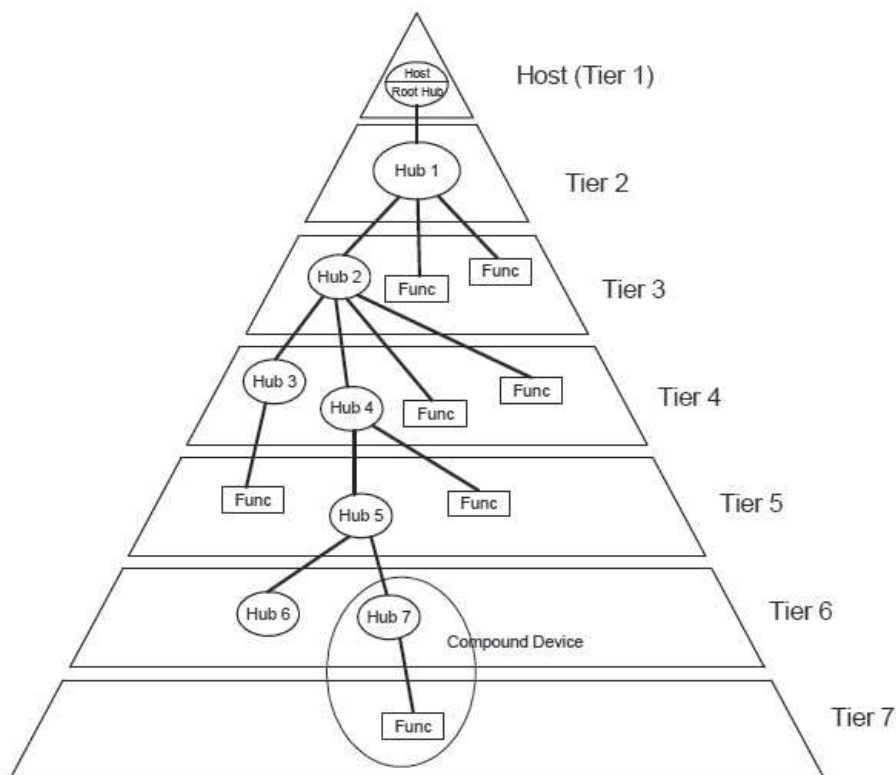
USB (abbreviation of **Universal Serial Bus**) was designed as a low cost, serial interface solution with bus power provided from the USB host to support a wide range of peripheral devices. The original bus speeds for USB were low speed of 1.5 Mbps, followed by full speed of 12 Mbps, and then high speed at 480 Mbps. With the advent of the USB 3.0 specification, the super speed was defined at 4.8 Gbps. Maximum data throughput, i.e. the line rate minus overhead is approximately 384 Kbps, 9.728 Mbps, and 425.984 Mbps for low, full and high speed respectively. Note that this is the maximum data throughput and it can be adversely affected by a variety of factors, including software processing, other USB bandwidth utilization on the same bus, etc.

One of the biggest advantages of USB is that it supports dynamic attachment and removal, which is a type of interface referred to as "plug and play". Following attachment of a USB peripheral device, the host and the device communicate to automatically advance the externally visible device state from the attached state through powered, default, addressed and finally to the configured states. Additionally, all devices must conform to the suspend state in which a very low bus power consumption specification must be met. Power conservation in the suspended state is another USB benefit.

Throughout this chapter, we will focus on the USB 2.0 specification, which includes low, full and high speed device specifications. Compliance to USB 2.0 specification for peripheral devices, does not necessarily indicate that the device is a high speed device, however a hub advertised as USB 2.0 compliant, must be high speed capable. A USB 2.0 device can be High Speed, Full Speed or Low Speed.

USB 2.0 bus topology

USB devices fall into the category of hubs - which provide additional downstream attachment points, or functions - which provide a capability to the system. The USB physical interconnection is a tiered star topology (see next Figure). Starting with the host and "root hub" at tier 1, up to seven tiers with a maximum of 127 devices can be supported. Tier 2 through 6 may have one or more hub devices in order to support communication to the next tier. A compound device (one which has both a hub and peripheral device functions) may not occupy tier 7.



The physical USB interconnection is accomplished for all USB 2.0 (up to high speed) devices via a simple 4-wire interface with bi-directional differential data (D+ and D-), power (VBUS) and ground. The VBUS power is nominally +5V. An "A-type" connector and mating plug are used for all host ports as well as downstream facing hub ports. A "B-type" connector and mating plug are used for all peripheral devices as well as the upstream facing port of a hub. Cable connections between host, hubs and devices can each be a maximum of 5 meters or ~16 feet. With the maximum of 7 tiers, cabling connections can be up to 30 meters or ~98 feet total.

USB bus enumeration and device layout

USB is a Host-controlled polled bus where all transactions are initiated by the USB host. Nothing on the bus happens without the host first initiating it; the USB devices cannot initiate a transaction, it is the host which polls each device, requesting data or sending data. All attached and removed USB devices are identified by a process termed "bus enumeration".

An attached device is recognized by the host and its speed (low, full or high) is identified via a signaling mechanism using the D+/D- USB data pair. When a new USB device is connected to the bus through a hub, the device enumeration process starts. Each hub provides an IN endpoint, which is used to inform the host about newly attached devices. The host continually polls on this endpoint to receive device attachment and removal events from the hub. Once a new device was attached, and the hub notified the host about this event, the USB bus driver of the host enables the attached device and starts requesting information from the device. This is done with standard USB requests which are sent through the default control pipe to endpoint zero of the device.

Information is requested in terms of **descriptors**. USB descriptors are data structures that are provided by devices to describe all of their attributes. This includes, e.g., the product/vendor ID, any device class affiliation and strings describing the product and vendor. Additionally, information about all available endpoints is provided. After the host reads all the necessary information from the device, it tries to find a matching device driver. The details of this process are dependant on the used operating system. After the first descriptors were read from the attached USB device, the host uses the vendor and product ID from the device descriptor to find a matching device driver.

The attached device will initially utilize the default USB address of 0. Additionally, all USB devices are comprised of a number of independent endpoints, which provide a terminus for communication flow between the host and device.

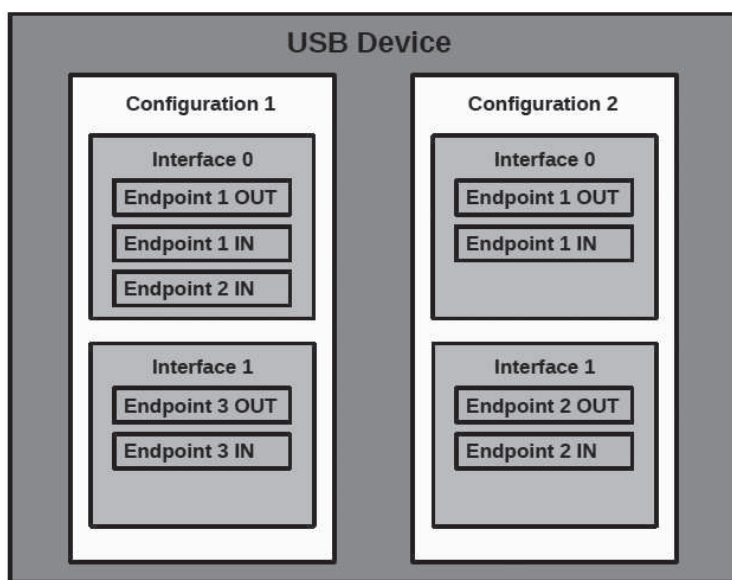
Endpoints can be categorized into **control** and **data** endpoints. Every USB device must provide at least one control endpoint at address 0 called the default endpoint or Endpoint0. This endpoint is bidirectional, that is, the host can send data to the endpoint and receive data from it within one transfer. The purpose of a control transfer is to enable the host to obtain device information, configure the device, or perform control operations that are unique to the device.

The endpoint is a buffer that typically consists of a block of memory or registers which stores received data or contain data which is ready to transmit. Each endpoint is assigned a unique endpoint number determined at design time, however, all devices must support the default control endpoint (ep0) which is assigned number 0 and may transfer data in both directions. All other endpoints may transfer data in one direction (always from the host perspective), labeled "Out", i.e., data from the host, or "In", i.e., data to the host. The endpoint number is a 4-bit integer associated with an endpoint (0-15); the same endpoint number is used to describe two endpoints, for example, EP 1 IN and EP 1 OUT. An endpoint address is the combination of an endpoint number and an endpoint direction, for example, EP 1 IN, EP 1 OUT and EP 3 IN. The endpoint addresses are encoded with the direction and number in a single byte, where the direction is the MSB (1=IN, 0=OUT) and the number is the lower four bits. For example:

- EP 1 IN = 0x81

- EP 1 OUT = 0x01
- EP 3 IN = 0x83
- EP 3 OUT = 0x03

A USB configuration defines the capabilities and features of a device, mainly its power capabilities and interfaces. The device can have multiple configurations, but only one is active at a time. A configuration can have one or more USB interfaces that define the functionality of the device. Typically, there is a one-to-one correlation between a function and an interface. However, certain devices expose multiple interfaces related to one function. For example, a USB device that comprises a keyboard with a built-in speaker will offer an interface for playing audio and an interface for key presses. In addition, the interface contains alternate settings that define the bandwidth requirements of the function associated with the interface. Each interface contains one or more endpoints, which are used to transfer data to and from the device. To sum up, a group of endpoints, form an interface, and a set of interfaces constitutes a configuration in the device. The following image shows a multiple-interfaces USB device:



After a matched device driver was loaded, it's the task of the device driver to select one of the provided device configurations, one or more interfaces within that configuration, and an alternate setting for each interface. Most USB devices don't provide multiple interfaces or multiple alternate settings. The device driver selects one of the configurations based on its own capabilities and the available bandwidth on

the bus and activates this configuration on the attached device. At this point, all interfaces and their endpoints of the selected configuration are set up and the device is ready for use.

Communication from the host to each device endpoint uses a communication "pipe" which is established during enumeration. The pipe is a logical association between the host and the device. The pipe is purely a software term. A pipe talks to an endpoint on a device, and that endpoint has an address. The other end of a pipe is always the host controller. A pipe for an endpoint is opened when the device is configured either by selecting a configuration and an interface's alternate setting. Therefore they become targets for I/O operations. A pipe has all the properties of an endpoint, but it is active and is used to communicate with the host. The combination of the device address, endpoint number and direction allows the host to uniquely reference each endpoint.

USB data transfers

Once the enumeration is complete, the host and device are free to carry out communications via data transfers from the host to the device or vice versa. Both directions of transfers are initiated by the host. Four different types of transfers are defined. These types are:

- **Control Transfers:** Used to configure a device at attach time and can be used for other device-specific purposes, for example, device specific register read/write access as well as control of other pipes on the device. Control transfers consist of up to three distinct stages: a setup stage containing a request, a data stage if necessary to or from the host and a status stage indicating the success of the transfer. USB has a number of standardized transactions that are implemented by using control transfers. For example, the "Set Address" and "Get Descriptor" transactions are always utilized in the device enumeration procedure described above. The "Set Configuration" request is another standard transaction which is also used during device enumeration.
- **Bulk Data Transfers:** Capable of transferring relatively large quantities of data or bursty data. Bulk transfers do not have guaranteed timing, but can provide the fastest data transfer rates if the USB bus is not occupied by other activity.
- **Interrupt Data Transfers:** Used for timely, but reliable delivery of data, for example, characters or coordinates with human-perceptible echo or feedback response characteristics. Interrupt transfers have a guaranteed maximum latency. USB mice and keyboards typically use interrupt data transfers.
- **Isochronous Data Transfers:** Occupy a pre-negotiated amount of USB bandwidth with a pre-negotiated delivery latency. Isochronous transfers have guaranteed timing, but do not have error correction capability. Isochronous data must be delivered at

the rate received to maintain its timing and additionally may be sensitive to delivery delays. A typical use for isochronous transfers would be for streaming audio or video.

USB device classes

The USB specification and supplemental documents define a number of device classes that categorize USB devices, according to capability and interface requirements. When a host retrieves device information, the class classification helps the host to determine how to communicate with the USB device. The hub is a specially designated class of devices that has additional requirements in the USB specification. Other examples of classes of peripheral devices are human interface, also known as HID, printer, imaging, mass storage and communications. The USB UART devices usually fall into the communications device class (CDC) of USB devices.

Human interface device class

The HID class devices usually interface with humans in some capacity. HID-class devices include mice, keyboards, printers, etc. However, the HID specification merely defines basic requirements for devices and the protocol for data transfer, and devices do not necessarily depend on any direct human interaction. HID devices must meet a few general requirements that are imposed to keep the HID interface standardized and efficient.

- All HID devices must have a control endpoint (Endpoint 0) and an interrupt IN endpoint. Many devices also use an interrupt OUT endpoint. In most cases, HID devices are not allowed to have more than one OUT and one IN endpoint.
- All data transferred must be formatted as reports whose structure is defined in the report descriptor.
- HID devices must respond to standard HID requests in addition to all standard USB requests.

Before the HID device can enter its normal operating mode and transfer data to the host, the device must properly enumerate. The enumeration process consists of a number of calls made by the host for descriptors stored in the device that describe the device's capabilities. The device must respond with descriptors that follow a standard format. Descriptors contain all basic information about a device. The USB specification defines some of the descriptors retrieved, and the HID specification defines other required descriptors. The following section discusses the descriptor structures a host expects to receive.

USB descriptors

The host software obtains descriptors from an attached device by sending various standard control requests to the default endpoint during enumeration, immediately upon a device being attached. Those requests specify the type of descriptor to retrieve. In response to such requests, the device sends descriptors that include information about the device, its configurations, interfaces and the related endpoints. Device descriptors contain information about the whole device.

Every USB device exposes a **device descriptor** that indicates the device's class information, vendor and product identifiers and number of configurations. Each configuration exposes its **configuration descriptor** that indicates the number of interfaces and power characteristics. Each interface exposes an **interface descriptor** for each of its alternate settings that contains information about the class and the number of endpoints. Each endpoint within each interface exposes **endpoint descriptors** that indicate the endpoint type and the maximum packet size.

Descriptors begin with a byte describing the descriptor length in bytes. This length equals the total number of bytes in the descriptor, including the byte storing the length. The next byte indicates the descriptor type, which allows the host to correctly interpret the rest of the bytes contained in the descriptor. The content and values of the rest of the bytes are specific to the type of descriptor being transmitted. The descriptor structure must follow specifications exactly; the host will ignore received descriptors containing errors in size or value, potentially causing enumeration to fail and prohibiting further communication between the device and the host.

USB device descriptors

Every Universal Serial Bus (USB) device must be able to provide a single device descriptor that contains relevant information about the device. For example, the **idVendor** and **idProduct** fields specify the vendor and product identifiers, respectively. The **bcdUSB** field indicates the version of the USB specification to which the device conforms. For example, 0x0200 indicates that the device is designed as per the USB 2.0 specification. The **bcdDevice** value indicates the device defined revision number. The device descriptor also indicates the total number of configurations that the device supports. You can see below an example of a structure that contains all the device descriptor fields:

```
typedef struct __attribute__((packed))
{
    uint8_t bLength;           // Length of this descriptor.
    uint8_t bDescriptorType;   // DEVICE descriptor type
    (USB_DESCRIPTOR_DEVICE).
    uint16_t bcdUSB;           // USB Spec Release Number (BCD).
    uint8_t bDeviceClass;      // Class code (assigned by the USB-IF).
    0xFF-Vendor specific.
    uint8_t bDeviceSubClass;   // Subclass code (assigned by the USB-IF).
```

```

    uint8_t bDeviceProtocol;        // Protocol code (assigned by the USB-IF).
0xFF-Vendor specific.
    uint8_t bMaxPacketSize0;        // Maximum packet size for endpoint 0.
    uint16_t idVendor;               // Vendor ID (assigned by the USB-IF).
    uint16_t idProduct;              // Product ID (assigned by the manufacturer).
    uint16_t bcdDevice;              // Device release number (BCD).
    uint8_t iManufacturer;           // Index of String Descriptor describing the manufacturer.
    uint8_t iProduct;                // Index of String Descriptor describing the product.
    uint8_t iSerialNumber;           // Index of String Descriptor with the device's serial
number.
    uint8_t bNumConfigurations;      // Number of possible configurations.

} USB_DEVICE_DESCRIPTOR

```

The first item **bLength** describes the descriptor length and should be common to all USB device descriptors.

The item **bDescriptorType** is the constant one-byte designator for device descriptors and should be common to all device descriptors.

The BCD-encoded two-byte **bcdUSB** item tells the system which USB specification release guidelines the device follows. This number might need to be altered in devices that take advantage of additions or changes included in future revisions of the USB specification, as the host will use this item to help determine what driver to load for the device.

If the USB device class is to be defined inside the device descriptor, this item **bDeviceClass** would contain a constant defined in the USB specification. Device classes defined in other descriptors should set the device class item in the device descriptor to 0x00.

If the device class item discussed above is set to 0x00, then the device **bDeviceSubClass** item should also be set to 0x00. This item can tell the host information about the device's subclass setting.

The item **bDeviceProtocol** can tell the host whether the device supports high-speed transfers. If the above two items are set to 0x00, this one should also be set to 0x00.

The item **bMaxPacketSize0** tells the host the maximum number of bytes that can be contained inside a single control endpoint transfer. For low-speed devices, this byte must be set to 8, while full-speed devices can have maximum endpoint 0 packet sizes of 8, 16, 32, or 64.

The two-byte item **idVendor** identifies the vendor ID for the device. Vendor IDs can be acquired through the USB.org website. Host applications will search attached USB devices' vendor IDs to find a particular device needed for an application.

Like the vendor ID, the two-byte item **idProduct** uniquely identifies the attached USB device. Product IDs can be acquired through the USB.org web site.

The item **bcdDevice** is used along with the vendor ID and the Product ID to uniquely identify each USB device.

The next three one-byte items tell the host which string array index to use when retrieving UNICODE strings describing attached devices that are displayed by the system on-screen. This string describes the manufacturer of the attached device. An **iManufacturer** string index value of 0x00 indicates to the host that the device does not have a value for this string stored in memory.

The index **iProduct** will be used when the host wants to retrieve the string that describes the attached product. For example the string could read "USB Keyboard".

The string pointed to by the index **iSerialNumber** can contain the UNICODE text for the product's serial number.

This item **bNumConfigurations** tells the host how many configurations the device supports. A configuration is the definition of the device's functional capabilities, including endpoint configuration. All devices must contain at least one configuration, but more than one can be supported.

USB configuration descriptor

The USB device can have several different configurations, although most of the devices only have one. The configuration descriptor specifies how the device is powered, its maximum power consumption and the number of its interfaces. There are two possible configurations, one for when the device is bus powered and another when it is mains powered. You can see below an example of a structure that contains all the configuration descriptor fields:

```
typedef struct __attribute__((packed))
{
    uint8_t bLength;           // Size of Descriptor in Bytes
    uint8_t bDescriptorType;   // Configuration Descriptor (0x02)
    uint16_t wTotalLength;     // Total length in bytes of data returned
    uint8_t bNumInterfaces;    // Number of Interfaces
    uint8_t bConfigurationValue; // Value to use as an argument to select this configuration
    uint8_t iConfiguration;    // Index of String Descriptor describing this configuration
    uint8_t bmAttributes;      // power parameters for the configuration
    uint8_t bMaxPower;         // Maximum Power Consumption in 2mA units
} USB_CONFIGURATION_DESCRIPTOR;
```

The item **bLength** defines the length of the configuration descriptor. This is a standard length.

The item **bDescriptorType** is the constant one-byte 0x02 designator for configuration descriptors.

The two-byte **wTotalLength** item defines the length of this descriptor and all of the other descriptors associated with this configuration. For example, the length could be calculated by adding the length of the configuration descriptor, the interface descriptor, the HID class descriptor, and two endpoint descriptors associated with this interface. This two-byte item follows a "little

endian" data format. The item defines the length of this descriptor and all of the other descriptors associated with this configuration.

The **bNumInterfaces** item defines the number of interface settings contained in this configuration.

The **bConfigurationValue** item is used by the SetConfiguration request to select this configuration.

The **iConfiguration** item is a index to a string descriptor describing the configuration in human readable form.

The **bmAttributes** item tells the host whether the device supports USB features such as remote wake-up. Item bits are set or cleared to describe these conditions. Check the USB specification for a detailed discussion on this item.

The **bMaxPower** item tells the host how much current the device will require to function properly at this configuration.

USB interface descriptor

The USB interface descriptor may contain information about alternate settings of a USB interface. The interface descriptor has a **bInterfaceNumber** field which specifies the interface number and a **bAlternateSetting** field which allows alternative settings for that interface. For example, you could have a device with two interfaces. The first interface could have a **bInterfaceNumber** set to zero, indicating it is the first interface descriptor and a **bAlternativeSetting** set to zero. The second interface could have a **bInterfaceNumber** set to one and a **bAlternativeSetting** set to zero (default). This second interface could also have a **bAlternativeSetting** set to one, being an alternative setting for the second interface.

The **bNumEndpoints** item provides the number of endpoints used by the interface.

The **bInterfaceClass**, **bInterfaceSubClass** and **bInterfaceProtocol** items specify the supported classes (HID, mass storage, etc.). This allows many devices to use class drivers preventing the need to write specific drivers for your device. The **iInterface** item allows for a string description of the interface.

You can see below an example of a structure containing the interface descriptor fields:

```
typedef struct __attribute__((packed))
{
    uint8_t bLength;           // Size of Descriptor in Bytes (9 Bytes)
    uint8_t bDescriptorType;   // Interface Descriptor (0x04)
    uint8_t bInterfaceNumber;  // Number of Interface
    uint8_t bAlternateSetting; // Value used to select alternative setting
    uint8_t bNumEndpoints;     // Number of Endpoints used for this interface
    uint8_t bInterfaceClass;    // Class Code (Assigned by USB Org)
    uint8_t bInterfaceSubClass; // Subclass Code (Assigned by USB Org)
    uint8_t bInterfaceProtocol; // Protocol Code (Assigned by USB Org)
```

```
uint8_t iInterface;           // Index of String Descriptor Describing this interface

} USB_INTERFACE_DESCRIPTOR;
```

USB endpoint descriptor

The USB endpoint descriptors describe endpoints which are different to endpoint zero. Endpoint zero is a control endpoint which is configured before any other descriptors. The host will use the information returned from these USB endpoint descriptors to specify the transfer type, direction, polling interval and maximum packet size for each endpoint. You can see below an example of a structure that contains all the endpoint descriptor fields:

```
typedef struct __attribute__((packed))
{
    uint8_t bLength;           // Size of Descriptor in Bytes (7 bytes)
    uint8_t bDescriptorType;    // Endpoint Descriptor (0x05)
    uint8_t bEndpointAddress;   // Endpoint Address.Bits 0..3b Endpoint Number. Bits 4..6b
Reserved.Set to Zero.Bits 7 Direction 0 = Out, 1 = In
    uint8_t bmAttributes        // Transfer type
    uint16_t wMaxPacketSize;     // Maximum Packet Size this endpoint can send or receive
    uint8_t bInterval;          // Interval for polling endpoint data transfers

} USB_ENDPOINT_DESCRIPTOR;
```

The **bEndpointAddress** indicates what endpoint this descriptor is describing.

The **bmAttributes** specifies the transfer type. This can either be Control, Interrupt, Isochronous or Bulk Transfers. If an Isochronous endpoint is specified, additional attributes can be selected such as the synchronisation and usage types. **Bits 0..1** are the transfer type: 00 = Control, 01 = Isochronous, 10 = Bulk, 11 = Interrupt. **Bits 2..7** are reserved. If the endpoint is Isochronous, **Bits 3..2** = Synchronisation Type (Iso Mode): 00 = No Synchronization, 01 = Asynchronous, 10 = Adaptive, 11 = Synchronous. **Bits 5..4** = Usage Type (Iso Mode): 00 = Data Endpoint, 01 = Feedback Endpoint, 10 = Explicit Feedback Data Endpoint, 11 = Reserved.

The **wMaxPacketSize** item indicates the maximum payload size for this endpoint.

The **bInterval** item is used to specify the polling interval of endpoint data transfers. Ignored for Bulk and Control Endpoints. The units are expressed in frames, thus this equates to either 1ms for low/full speed devices and 125us for high speed devices.

USB string descriptors

The USB string descriptors (USB_STRING_DESCRIPTOR) provide human readable information to the other descriptors. They are optional. If a device does not support string descriptors, all references to string descriptors within device, configuration, and interface descriptors must be set to zero.

String descriptors are UNICODE encoded characters so that multiple languages can be supported with a single product. When requesting a string descriptor, the requester specifies the desired language using a 16-bit language ID (LANGID) defined by the USB-IF. String index zero is used for all languages and returns a string descriptor that contains an array of two-byte LANGID codes supported by the device.

Offset	Field	Type	Size	Value	Description
0	bLength	uint8_t	N + 2	Number	Size of this descriptor in bytes.
1	bDescriptorType	uint8_t	1	Constant	String Descriptor Type
2	wLANGID[0]	uint8_t	2	Number	LANGID code zero (for example 0x0407 German (Standard)).
...	...	...	...	...	...
N	wLANGID[x]	uint8_t	2	Number	LANGID code zero x (for example 0x0409 English (United States)).

The UNICODE string descriptor is not NULL-terminated. The string length is computed by subtracting two from the value of the first byte of the descriptor.

Offset	Field	Type	Size	Value	Description
0	bLength	uint8_t	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	uint8_t	1	Constant	String Descriptor Type
2	bString	uint8_t	N	Number	UNICODE encoded string.

USB HID descriptor

The USB HID device class supports devices that are used by humans to control the operation of computer systems. The HID class of devices include a wide variety of human interface, data indicator and data feedback devices with various types of output directed to the end user. Some common examples of HID class devices include:

- Keyboards.
- Pointing devices such as a standard mouse, joysticks and trackballs.
- Front-panel controls like knobs, switches, buttons and sliders.
- Controls found on telephony, gaming or simulation devices such as steering wheels, rudder pedals and dial pads.
- Data devices such as bar-code scanners, thermometers and analyzers.

The following descriptors are required in a USB HID device:

- Standard device descriptor.
- Standard configuration descriptor.
- Standard interface descriptor for the HID class.
- Class-specific HID descriptor.
- Standard endpoint descriptor for interrupt IN endpoint.
- Class-specific report descriptor.

The class-specific HID descriptor looks like this:

```
typedef struct __attribute__((packed))
{
    uint8_t      bLength;
    uint8_t      bDescriptorType;
    uint16_t     bcdHID;
    uint8_t      bCountryCode;
    uint8_t      bNumDescriptors;
    uint8_t      bReportDescriptorType;
    uint16_t     wItemLength;
} USB_HID_DESCRIPTOR;
```

The **bLength** item describes the size of the HID descriptor. It can vary depending on the number of subordinate descriptors, such as report descriptors, that are included in this HID configuration definition.

The **bDescriptorType** 0x21 value is the constant one-byte designator for device descriptors and should be common to all HID descriptors.

The two-byte **bcdHID** item tells the host which version of the HID class specification the device follows. USB specification requires that this value be formatted as a binary coded decimal digit, meaning that the upper and lower nibbles of each byte represent the number 0...9. For example, 0x0101 represents the number 0101, which equals a revision number of 1.01 with an implied decimal point.

If the device was designed to be localized to a specific country, the **bCountryCode** item tells the host which country. Setting the item to 0x00 tells the host that the device was not designed to be localized to any country.

The **bNumDescriptors** item tells the host how many report descriptors are contained in this HID configuration. The following two-byte pairs of items describe each contained report descriptor.

The **bReportDescriptorType** item describes the first descriptor which will follow the transfer of this HID descriptor. For example, if the value is "0x22", indicates that the descriptor to follow is a report descriptor.

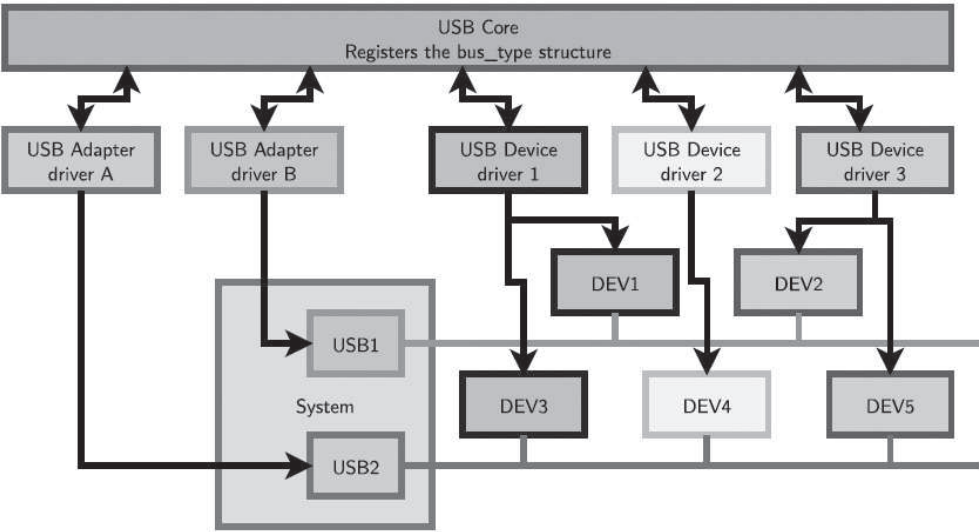
The **wItemLength** item tells the host the size of the descriptor that is described in the preceding item.

The **HID report descriptor** is a hard coded array of bytes that describe the device's data packets. This includes: how many packets the device supports, how large are the packets and the purpose of each byte and bit in the packet. For example, a keyboard with a calculator program button can tell the host that the button's pressed/released state is stored as the 2nd bit in the 6th byte in data packet number 4.

The Linux USB subsystem

USB support was added to Linux early in the 2.2 kernel series and has continued to develop since then. Besides support for each new generation of USB, various host controllers gained support, new drivers for peripherals have been added and advanced features for latency measurement and improved power management introduced.

In Linux, the "USB Core" is a specific API implemented to support USB peripheral devices and host controllers. This API abstracts all hardware by defining a set of data structures, macros and functions. Host-side drivers for USB devices talk to these "usbcore" APIs. There are two sets of APIs, one is intended for general-purpose USB device drivers (the ones that will be developed through this chapter), and the other is for drivers that are part of the core. Such core drivers include the hub driver, which manages trees of USB devices, and several different kinds of USB host adapter drivers, which control individual busses. The following image shows an example of a Linux USB Subsystem:



The Linux USB API supports synchronous calls for control and bulk messages. It also supports asynchronous calls for all kinds of data transfer by using request structures called "URBs" (USB Request Blocks).

The only host-side drivers that actually touch hardware (reading/writing registers, handling IRQs and so on) are the Host Controller Devices (HCDs) drivers. In theory, all HCDs provide the same functionality through the same API. In practice, that's becoming more true, but there are still differences that crop up, especially with fault handling on the less common controllers. Different controllers don't necessarily report the same aspects of failures, and recovery from faults (including software-induced ones like to unlink an URB) isn't yet fully consistent.

The main focus of this chapter is the development of Linux USB device drivers. All the sections that follow are related to the development of this type of drivers.

Writing Linux USB device drivers

In the following labs, you will develop several Linux drivers that will help you to understand the basic framework of a Linux USB device driver. Before you proceed with the labs, it will be explained the main Linux USB data structures and functions in the following sections.

USB device driver registration

The first thing a Linux USB device driver needs to do is register itself with the Linux USB core, giving it some information about which devices the driver supports and which functions to call when a device supported by the driver is inserted or removed from the system. All of this information is passed to the USB core in the `usb_driver` structure. See below the `usb_driver` definition for a USB seven segment driver located at `drivers/usb/misc/usbsevseg.c` in the kernel source tree:

```
static struct usb_driver sevseg_driver = {
    .name =          "usbsevseg",
    .probe =         sevseg_probe,
    .disconnect =    sevseg_disconnect,
    .suspend =       sevseg_suspend,
    .resume =        sevseg_resume,
    .reset_resume =  sevseg_reset_resume,
    .id_table =      id_table,
};
```

The variable name is a string that describes the driver. It is used in informational messages printed to the system log. The `probe()` and `disconnect()` hotplugging callbacks are called when a device that matches the information provided in the `id_table` variable is either seen or removed.

The `probe()` function is called by the USB core into the driver to see if the driver is willing to manage a particular interface on a device. If it is, `probe()` returns zero and uses `usb_set_intfdata()` to associate driver specific data with the interface. It may also use `usb_set_interface()` to specify the appropriate altsetting. If unwilling to manage the interface, return `-ENODEV`, if genuine IO errors occurred, an appropriate negative `errno` value.

```
int (* probe) (struct usb_interface *intf, const struct usb_device_id *id);
```

The `disconnect()` callback is called when the interface is no longer accessible, usually because its device has been (or is being) disconnected, or the driver module is being unloaded.

```
void disconnect(struct usb_device *dev, void *drv context);
```

In the `usb_driver` structure, some power management (PM) callbacks are defined:

- **suspend:** Called when the device is going to be suspended.
- **resume:** Called when the device is being resumed.
- **reset_resume:** Called when the suspended device has been reset instead of being resumed.

There are also defined some device level operations in the `usb_driver` structure:

- **pre_reset:** Called when the device is about to be reset.
- **post_reset:** Called after the device has been reset.

The USB device drivers use an ID table to support hotplugging. The pointer variable `id_table`, which is included in the `usb_driver` structure, points to an array of structures of type `usb_device_id` that announce the devices that the USB device driver supports. Most drivers use the `USB_DEVICE()` macro to create `usb_device_id` structures. These structures are registered to the USB core by using the `MODULE_DEVICE_TABLE(usb, xxx)` macro. The following lines of code, included in the `drivers/usb/misc/usbsevseg.c` driver, create and register a USB device to the USB core:

```
#define VENDOR_ID    0x0fc5
#define PRODUCT_ID   0x1227

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
    { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
    { },
};
MODULE_DEVICE_TABLE(usb, id_table);
```

The `usb_driver` structure is registered to the bus core by using the `module_usb_driver()` function:

```
module_usb_driver(sevseg_driver);
```

Linux host-side data types

USB device drivers actually bind to interfaces, not devices. Think of them as "interface drivers", though you may not see many devices where the distinction is important. Most USB devices are simple, with only one function, one configuration, one interface and one alternate setting. The USB interface is represented by the `usb_interface` structure. This is what the USB core passes to the USB driver's `probe()` function when this callback function is being called.

```
struct usb_interface {
    struct usb_host_interface *altsetting;
    struct usb_host_interface *cur_altsetting;
    unsigned num_altsetting;
    struct usb_interface_assoc_descriptor *intf_assoc;
    int minor;
    enum usb_interface_condition condition;
    unsigned sysfs_files_created:1;
    unsigned ep_devs_created:1;
    unsigned unregistering:1;
    unsigned needs_remote_wakeup:1;
    unsigned needs_altsetting0:1;
    unsigned needs_binding:1;
    unsigned resetting_device:1;
    unsigned authorized:1;
    struct device dev;
    struct device *usb_dev;
    atomic_t pm_usage_cnt;
    struct work_struct reset_ws;
};
```

These are the main members of the `usb_interface` structure:

- **altsetting**: Array of `usb_host_interface` structures, one for each alternate setting that may be selected. Each one includes a set of endpoint configurations. They will be in no particular order. The `usb_host_interface` structure for each alternate setting allows to access the `usb_endpoint_descriptor` structure for each of its endpoints.
- **cur_altsetting**: The current altsetting.
- **num_altsetting**: Number of altsettings defined.

Each interface may have alternate settings. The initial configuration of a device sets altsetting 0, but the device driver can change that setting by using `usb_set_interface()`. Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-conformant USB devices that use isochronous endpoints will use them in non-default settings.

The `usb_host_interface` structure includes an array of `usb_host_endpoint` structure:

```
/* host-side wrapper for one interface setting's parsed descriptors */
struct usb_host_interface {
    struct usb_interface_descriptor desc;
    int extralen;
    unsigned char *extra; /* Extra descriptors */

    /*
     * array of desc.bNumEndpoints endpoints associated with this
     * interface setting. These will be in no particular order.
     */
    struct usb_host_endpoint *endpoint;
    char *string; /* iInterface string, if present */
};
```

Each `usb_host_endpoint` structure includes a `usb_endpoint_descriptor` structure:

```
struct usb_host_endpoint {
    struct usb_endpoint_descriptor desc;
    struct usb_ss_ep_comp_descriptor ss_ep_comp;
    struct usb_ssp_isoc_ep_comp_descriptor ssp_isoc_ep_comp;
    struct list_head urb_list;
    void *hcpriv;
    struct ep_device *ep_dev; /* For sysfs info */

    unsigned char *extra; /* Extra descriptors */
    int extralen;
    int enabled;
    int streams;
};
```

The `usb_endpoint_descriptor` structure contains all the USB-specific data announced by the device itself:

```
struct usb_endpoint_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __u8  bEndpointAddress;
    __u8  bmAttributes;
    __le16 wMaxPacketSize;
    __u8  bInterval;

    /* NOTE: these two are _only_ in audio endpoints. */
    /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
    __u8  bRefresh;
    __u8  bSynchAddress;
} __attribute__((packed));
```

You can use the following code snippet to obtain the IN and OUT endpoint addresses from the IN and OUT endpoint descriptors which are included in the current altsetting of the USB interface:

```
struct usb_host_interface *altsetting = intf->cur_altsetting;
int ep_in, ep_out;

/* There are two usb_host_endpoint structures in this interface altsetting. Each usb_host_
endpoint structure contains a usb_endpoint_descriptor */
ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
ep_out = altsetting->endpoint[1].desc.bEndpointAddress;
```

USB request block (URB)

Any communication between the host and device is done asynchronously by using USB Request Blocks (urbs):

- An URB consists of all relevant information to execute any USB transaction and deliver the data and status back.
- Execution of an URB is inherently an asynchronous operation, i.e., the `usb_submit_urb()` call returns immediately after it has successfully queued the requested action.
- Transfers for one URB can be canceled by calling `usb_unlink_urb()` at any time.
- Each URB has a completion handler which is called after the action has been successfully completed or canceled. The URB also contains a context-pointer for passing information to the completion handler.
- Each endpoint for a device logically supports a queue of requests. You can fill that queue so that the USB hardware can still transfer data to an endpoint while your driver handles the completion of another. This maximizes use of USB bandwidth and

supports seamless streaming of data to (or from) devices when using periodic transfer modes.

These are some important fields of the urb structure:

```
struct urb
{
    // (IN) device and pipe specify the endpoint queue
    struct usb_device *dev;           // pointer to associated USB device
    unsigned int pipe;               // endpoint information

    unsigned int transfer_flags;      // URB_ISO_ASAP, URB_SHORT_NOT_OK, etc.

    // (IN) all urbs need completion routines
    void *context;                   // context for completion routine
    usb_complete_t complete;         // pointer to completion routine

    // (OUT) status after each completion
    int status;                      // returned status

    // (IN) buffer used for data transfers
    void *transfer_buffer;           // associated data buffer
    u32 transfer_buffer_length;      // data buffer length
    int number_of_packets;           // size of iso_frame_desc

    // (OUT) sometimes only part of CTRL/BULK/INTR transfer_buffer is used
    u32 actual_length;               // actual data buffer length
    // (IN) setup stage for CTRL (pass a struct usb_ctrlrequest)
    unsigned char *setup_packet;     // setup packet (control only)

    // Only for PERIODIC transfers (ISO, INTERRUPT)
    // (IN/OUT) start_frame is set unless URB_ISO_ASAP isn't set
    int start_frame;                 // start frame
    int interval;                   // polling interval

    // ISO only: packets are only "best effort"; each can have errors
    int error_count;                 // number of errors
    struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

The USB driver must create a "pipe" using values from the appropriate endpoint descriptor in an interface that it's claimed.

URBs are allocated by calling `usb_alloc_urb()`:

```
struct urb *usb_alloc_urb(int isoframes, int mem_flags);
```

The return value is a pointer to the allocated URB, 0 if allocation failed. The parameter `isoframes` specifies the number of isochronous transfer frames you want to schedule. For CTRL/BULK/INT, use 0. The parameter `mem_flags` holds standard memory allocation flags, letting you control (among other things) whether the underlying code may block or not.

To free an URB, use `usb_free_urb()`:

```
void usb_free_urb(struct urb *urb);
```

Interrupt transfers are periodic and happen in intervals that are powers of two (1, 2, 4 etc) units. Units are frames for full and low speed devices and microframes for high speed ones. You can use the `usb_fill_int_urb()` macro to fill INT transfer fields. When the write urb is filled up with the proper information by using the `usb_fill_int_urb()` function, you should point the urb's completion callback to call your own callback function. This function is called when the urb is finished by the USB subsystem. The callback function is called in interrupt context, so caution must be taken not to do very much processing at that time. The `usb_submit_urb()` call modifies `urb->interval` to the implemented interval value that is less than or equal to the requested interval value.

An URB is submitted by using the function `usb_submit_urb()`:

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

The parameter `mem_flags`, such as `GFP_ATOMIC`, controls memory allocation, such as whether the lower levels may block when memory is tight. It immediately returns, either with status 0 (request queued) or some error code, usually caused by the following:

- Out of memory (-ENOMEM).
- Unplugged device (-ENODEV).
- Stalled endpoint (-EPIPE).
- Too many queued ISO transfers (-EAGAIN).
- Too many requested ISO frames (-EFBIG).
- Invalid INT interval (-EINVAL).
- More than one packet for INT (-EINVAL).

After submission, `urb->status` is `-EINPROGRESS`; however, you should never look at that value except in your completion callback.

There are two ways to cancel an URB you've submitted but which hasn't been returned to your driver yet. For an asynchronous cancel, call `usb_unlink_urb()`:

```
int usb_unlink_urb(struct urb *urb);
```

The `usb_unlink_urb()` function removes the urb from the internal list and frees all allocated HW descriptors. The status is changed to reflect unlinking. Note that the URB will not normally have finished when `usb_unlink_urb()` returns; you must still wait for the completion handler to be called.

To cancel an URB synchronously, call `usb_kill_urb()`:

```
void usb_kill_urb(struct urb *urb);
```

The `usb_kill_urb()` function does everything `usb_unlink_urb()` does, and in addition it waits until after the URB has been returned and the completion handler has finished.

The completion handler is of the following type:

```
typedef void (*usb_complete_t)(struct urb *);
```

In the completion handler, you should have a look at `urb->status` to detect any USB errors. Since the context parameter is included in the URB, you can pass information to the completion handler.

LAB 13.1: USB HID device application

In this first USB lab, you will learn how to create a fully functional USB HID device and how to send and receive data by using HID reports. For this lab, you are going to use the Curiosity PIC32MX470 Development Board:

<https://www.microchip.com/DevelopmentTools/ProductDetails/dm320103#additional-summary>

The Curiosity PIC32 MX470 Development Board features PIC32MX Series (PIC32MX470512H) with a 120MHz CPU, 512KB Flash, 128KB RAM , full Speed USB and multiple expansion options.

The Curiosity Development Board includes an integrated programmer/debugger, excellent user experience options with Multiple LED's, RGB LED and a switch. Each board provides two MikroBus® expansion sockets from MicroElektronika , I/O expansion header and a Microchip X32 header, which enable customers seeking accelerated application prototype development. The board is fully integrated with Microchip's MPLAB® X IDE and into PIC32's powerful software framework, MPLAB ® Harmony that provides flexible and modular interface for application development, a rich set of inter-operable software stack (TCP-IP, USB) and easy to use features.

The applications of this chapter have been developed using the Windows versions of the tools and are included in the GitHub of this book in the `PIC32MX_usb_labs.zip` file, which is included in the `linux_5.4_USB_drivers` folder. See below the used versions of the tools:

- **Development Environment:** MPLAB® X IDE v5.10
- **C Compiler:** MPLAB® XC32 v2.15
- **Software Tools:** MPLAB® Harmony Integrated Software Framework v2.06.
GenericHIDSimpleDemo application ("hid_basic" example of Harmony)

You can download all the previous SW versions from the following links:

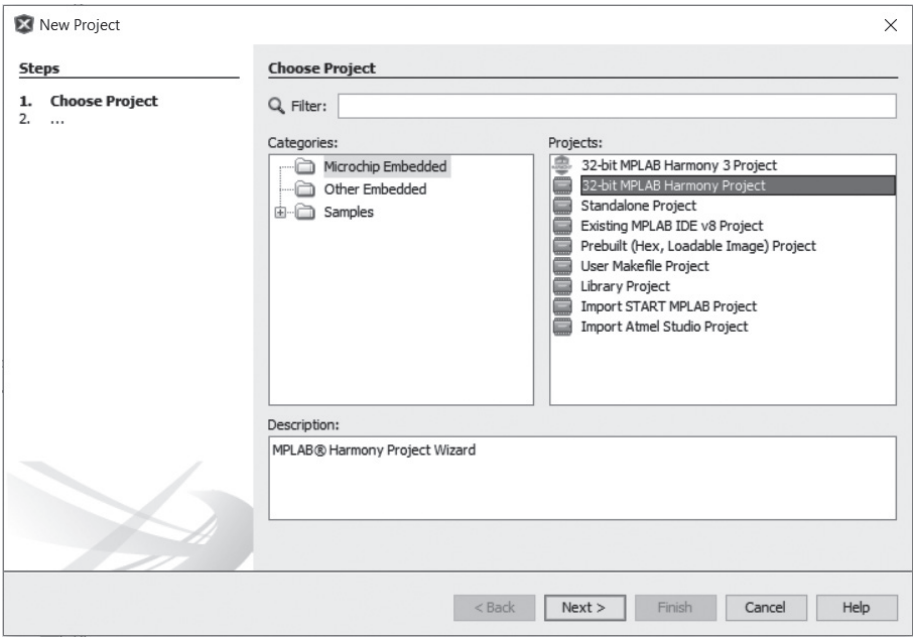
<https://www.microchip.com/development-tools/pic-and-dspic-downloads-archive>

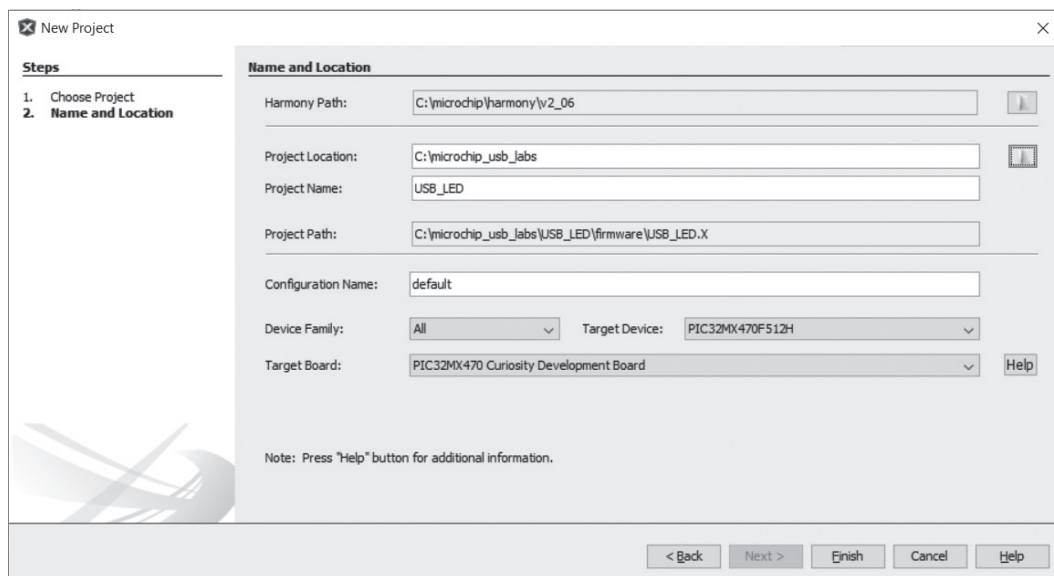
<https://www.microchip.com/en-us/development-tools-tools-and-software/embedded-software-center/mplab-harmony-v3/mplab-harmony-v2>

In this lab, you will use the MPLAB® Harmony Configurator Tool to create a USB Device that can be enumerated as a HID device and communicate with the Linux USB driver that you will develop in the following lab. These are the needed steps to create the USB device:

STEP 1: Create a new project

Create an empty 32-bit MPLAB Harmony Project, named USB_LED, for the Curiosity development board. Save the project in the following folder that was previously created: C:\microchip_usb_labs.

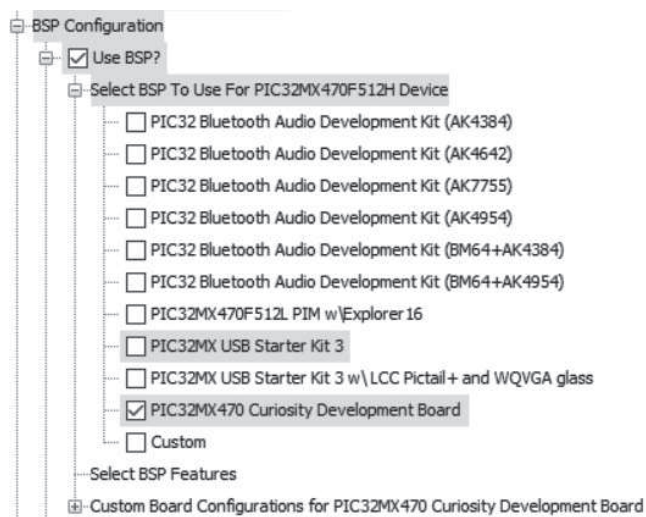




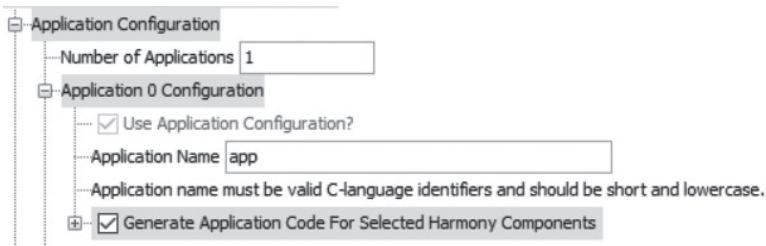
STEP 2: Configure Harmony

Launch the MPLAB Harmony Configurator plugin and click on Tools->Embedded->MPLAB Harmony Configurator.

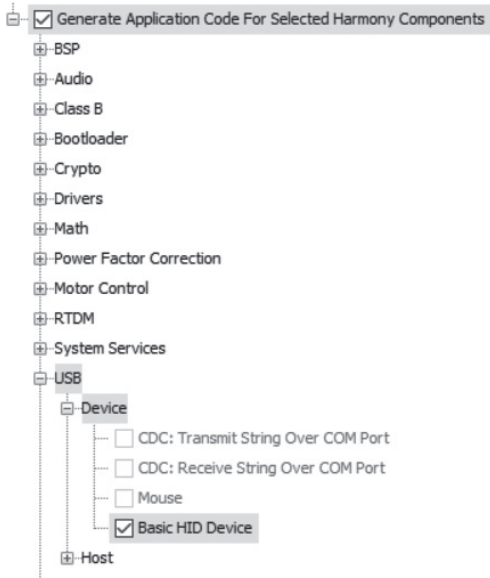
Select your demo board, enabling the BSP (Board Support Package):



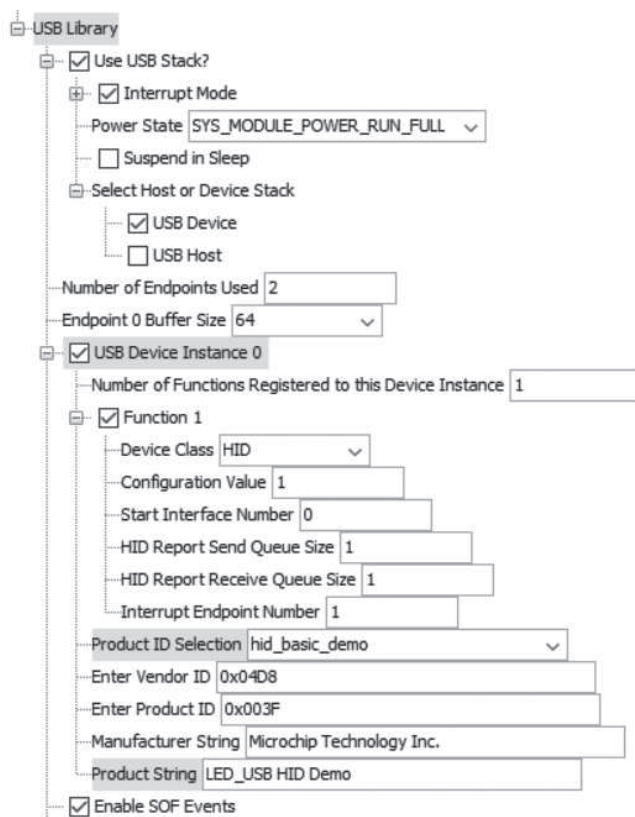
Enable the Generate Application Code For Selected Harmony Components:



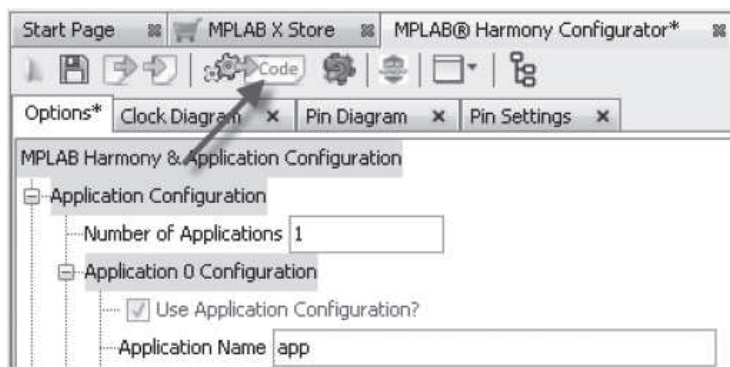
Select the Basic HID Device demo template:



At the USB Library option of Harmony Framework Configuration, select hid_basic_demo as Product ID Selection. Select also the Vendor ID, Product ID, Manufacturer String and Product String as shown in the following screen capture. You have to select a USB Device stack. The USB device will have one control endpoint (ep0) and one interrupt endpoint (composed of IN and OUT endpoints), so you will have to write the value two in the Number of Endpoints Used field. There will be only one configuration and one interface associated with the device.



Generate the code, save the modified configuration, and generate the project:



STEP 3: Modify the generated code

Typically, the HID class is used to implement human interface products, such as mice and keyboards. The HID protocol is however quite flexible and can be adapted and used to send/receive general purpose data to/from a USB device.

In the following two labs, you will see how to use the USB protocol for basic general purpose USB data transfer. You will develop Linux USB drivers that send USB commands to the PIC32MX USB HID device to toggle three LEDs (LED1, LED2, LED3) included in the PIC32MX Curiosity board. The PIC32MX USB HID device will also check the value of the user button (S1) and reply to the Raspberry Pi with a packet that contains its value.

In this lab, you have to implement the following in the USB device side:

- **Toggle LED(s):** The Linux USB driver sends a report to the HID device. The first byte of the report can be 0x01, 0x02, or 0x03. The HID device must toggle LED1 when it receives 0x01, LED2 when it receives 0x02 and LED3 when it receives 0x03 in the report.
- **Get Pushbutton State:** The Linux USB driver sends a report to the HID Device. The first byte of this report is 0x00. The HID device must reply with another report, where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed).

By examining the `app.c` code generated by the MPLAB Harmony Configurator, the template is expecting you to implement your USB State Machine inside the Function `USB_Task()`. The state machine will be executed if the HID device is configured; if the HID device is de-configured, the USB State Machine needs to return to the INIT state.

After initialization, the HID device needs to wait for a command from the host; scheduling a read request will enable the HID device to receive a report. The state machine needs to wait for the host to send the report. After receiving the report, the application needs to check the first byte of the report. If this byte is 0x01, 0x02 or 0x03, then LED1, LED2 and LED3 must be toggled. If the first byte is 0x00, a response report with the switch status must be sent to the host, then a new read must be scheduled.

STEP 4: Declare the USB State Machine states

To create a USB State Machine, you need to declare an enumeration type (e.g., `USB_STATES`) which contains the labels for the four states, needed to implement the state machine. (e.g., `USB_STATE_INIT`, `USB_STATE_WAITING_FOR_DATA`, `USB_STATE_SCHEDULE_READ`, `USB_STATE_SEND_REPORT`). Find the section Type Definitions in `app.h` file and declare the enumeration type:

```

typedef enum
{
    /* Application's state machine's initial state. */
    APP_STATE_INIT=0,
    APP_STATE_SERVICE_TASKS,

    /* TODO: Define states used by the application state machine. */

} APP_STATES;

/* Declare the USB State Machine states */
typedef enum
{
    /* Application's state machine's initial state. */
    USB_STATE_INIT=0,
    USB_STATE_WAITING_FOR_DATA,
    USB_STATE_SCHEDULE_READ,
    USB_STATE_SEND_REPORT

} USB_STATES;

```

STEP 5: Add new members to APP_DATA type

The APP_DATA structure type already contains members needed for the application state machine and the enumeration process (state, handleUsbDevice, usbDeviceIsConfigured, etc.). You need to add the members you will use to send and receive HID reports.

Find the APP_DATA structure type in app.h file, and add the following members:

- A member to store the USB State Machine status.
- Two pointers to buffer (one for data received and one for data to send).
- Two HID transfer handles (one for reception transfer and one for the transmission transfer).
- Two flags to indicate the state of the ongoing transfer (one for reception and one for transmission transfer).

```

typedef struct
{
    /* The application's current state */
    APP_STATES state;

    /* TODO: Define any additional data used by the application. */

    /*
     * USB variables used by the HID device application:
     *
     *     handleUsbDevice          : USB Device driver handle

```



```

    *      usbDeviceIsConfigured      : If true, USB Device is configured
    *      activeProtocol              : USB HID active Protocol
    *      idleRate                    : USB HID current Idle
    */
USB_DEVICE_HANDLE      handleUsbDevice;
bool                   usbDeviceIsConfigured;
uint8_t                activeProtocol;
uint8_t                idleRate;

/* Add new members to APP_DATA type */
/* USB_Task's current state */
USB_STATES stateUSB;

/* Receive data buffer */
uint8_t * receiveDataBuffer;

/* Transmit data buffer */
uint8_t * transmitDataBuffer;

/* Send report transfer handle*/
USB_DEVICE_HID_TRANSFER_HANDLE txTransferHandle;

/* Receive report transfer handle */
USB_DEVICE_HID_TRANSFER_HANDLE rxTransferHandle;

/* HID data received flag*/
bool hidDataReceived;

/* HID data transmitted flag */
bool hidDataTransmitted;

} APP_DATA;

```

STEP 6: Declare the reception and transmission buffers

To schedule a report receive or a report send request, you need to provide a pointer to a buffer to store the received data and the data that has to be transmitted. Find the section Global Data Definitions in app.c file and declare two 64 byte buffers:

```

APP_DATA appData;

/* Declare the reception and transmission buffers */
uint8_t receiveDataBuffer[64] __attribute__((aligned(16)));
uint8_t transmitDataBuffer[64] __attribute__((aligned(16)));

```

STEP 7: Initialize the new members

In Step 5, you added some new members to APP_DATA structure type; those members need to be initialized, and some of them need to be initialized just once in the APP_Initialize() function.

Find the `APP_Initialize()` function in `app.c` file, and add the code to initialize the USB State Machine state member and the two buffer pointers. The state variable needs to be set to the initial state of the USB State Machine. The two pointers need to point to the corresponding buffers you declared in Step 6. The other members will be initialized just before their use.

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;

    /* Initialize USB HID Device application data */
    appData.handleUsbDevice = USB_DEVICE_HANDLE_INVALID;
    appData.usbDeviceIsConfigured = false;
    appData.idleRate = 0;

    /* Initialize USB Task State Machine appData members */
    appData.receiveDataBuffer = &receiveDataBuffer[0];
    appData.transmitDataBuffer = &transmitDataBuffer[0];
    appData.stateUSB = USB_STATE_INIT;
}
```

STEP 8: Handle the detach

In the Harmony version we are using, `USB_DEVICE_EVENT_DECONFIGURED` and `USB_DEVICE_EVENT_RESET` events are not passed to the Application USB Device Event Handler Function, so the `usbDevicesConfigured` flag of `appData` structure needs to be set as false inside the `USB_DEVICE_EVENT_POWER_REMOVED` event.

Find the power removed case (`USB_DEVICE_EVENT_POWER_REMOVED`) in the `APP_USBDeviceEventHandler()` function (included in the `app.c` file), and set the member `usbDevicesConfigured` of `appData` structure to false:

```
case USB_DEVICE_EVENT_POWER_REMOVED:
    /* VBUS is not available any more. Detach the device. */
    /* STEP 8: Handle the detach */
    USB_DEVICE_Detach(appData.handleUsbDevice);
    appData.usbDeviceIsConfigured = false;
    /* This is reached from Host to Device */
    break;
```

STEP 9: Handle the HID events

The two flags you declared in Step 5 will be used by the USB State Machine to check the status of the previous report receive or transmit transaction. The status of those two flags need to be updated when the two HID events (report sent and report received) are passed to the Application HID Event Handler Function. You need to be sure that the event is related to the request you

made; for this purpose, you can compare the transfer handle of the request with the transfer handle available in the event. If they match, the event is related to the ongoing request.

Find the `APP_USBDeviceHIDEventHandler()` function in the `app.c` file, add a local variable to cast the `eventData` parameter, and update the two flags, one in the report received event and one in the report sent event; don't forget to check if the transfer handles are matched before setting the flag to true. To match the transfer handle, you need to cast the `eventData` parameter to the USB Device HID Report Event Data Type. There are two events and two types, one for report received and one for report sent.

```
static void APP_USBDeviceHIDEventHandler
(
    USB_DEVICE_HID_INDEX hidInstance,
    USB_DEVICE_HID_EVENT event,
    void * eventData,
    uintptr_t userData
)
{
    APP_DATA * appData = (APP_DATA *)userData;

    switch(event)
    {
        case USB_DEVICE_HID_EVENT_REPORT_SENT:
        {
            /*
             * This means a Report has been sent. We are free to send next
             * report. An application flag can be updated here.
             */

            /* Handle the HID Report Sent event */
            USB_DEVICE_HID_EVENT_DATA_REPORT_SENT * report =
                (USB_DEVICE_HID_EVENT_DATA_REPORT_SENT *)eventData;
            if(report->handle == appData->txTransferHandle)
            {
                // Transfer progressed.
                appData->hidDataTransmitted = true;
            }
            break;
        }
        case USB_DEVICE_HID_EVENT_REPORT_RECEIVED:
        {
            /* This means Report has been received from the Host. Report
             * received can be over Interrupt OUT or Control endpoint based on
             * Interrupt OUT endpoint availability. An application flag can be
             * updated here.
             */

            /* Handle the HID Report Received event */
            USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED * report =
                (USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED *)eventData;
            if(report->handle == appData->rxTransferHandle)
```

```

        {
            // Transfer progressed.
            appData->hidDataReceived = true;
        }
        break;
    }
    [...]
}

```

STEP 10: Create the USB State Machine

The Basic HID Device template that was used to generate the code expects the USB State machine to be placed inside the `USB_Task()` function. That state machine will be executed until the `usbDevicesConfigured` member of `appData` structure is true.

When the USB cable is unplugged, the state machine is no longer executed, but you need to reset it to the initial state to be ready for the next USB connection.

Find the `if(appData.usbDevicesConfigured)` statement in the `USB_Task()` function of the `app.c` file, and add the `else` statement to set the USB State Machine state member of the `appData` structure to its initial state (`USB_STATE_INIT`).

Inside the `if` statement of the `USB_Task()` function, you can place the requested state machine. You can create it using a `switch` statement with four cases, one for each state you declared in the enumeration type you defined in Step 4. Find the `if(appData.usbDevicesConfigured)` statement in the `USB_Task()` function, and add a `switch` statement for the USB State Machine state member of the `appData` structure and a case for each entry of the enumeration type of that state member.

Inside the initialization state of the `switch` statement, add the code to set the transmission flag to true and the two transfer handles to invalid (`USB_DEVICE_HID_TRANSFER_HANDLE_INVALID`), and set the USB State Machine state member of `appData` structure to the state that schedules a receive request (`USB_STATE_SCHEDULE_READ`):

```

static void USB_Task (void)
{
    if(appData.usbDeviceIsConfigured)
    {
        /*
         * Write USB HID Application Logic here. Note that this function is
         * being called periodically the APP_Tasks() function. The application
         * logic should be implemented as state machine. It should not block
         */

        switch (appData.stateUSB)
        {
            case USB_STATE_INIT:

```

```

    appData.hidDataTransmitted = true;
    appData.txTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
    appData.rxTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
    appData.stateUSB = USB_STATE_SCHEDULE_READ;

    break;

case USB_STATE_SCHEDULE_READ:

    appData.hidDataReceived = false;
    USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
        &appData.rxTransferHandle, appData.receiveDataBuffer, 64);
    appData.stateUSB = USB_STATE_WAITING_FOR_DATA;

    break;

case USB_STATE_WAITING_FOR_DATA:

    if( appData.hidDataReceived )
    {
        if (appData.receiveDataBuffer[0]==0x01)
        {
            BSP_LED_1Toggle();
            appData.stateUSB = USB_STATE_SCHEDULE_READ;
        }
        else if (appData.receiveDataBuffer[0]==0x02)
        {
            BSP_LED_2Toggle();
            appData.stateUSB = USB_STATE_SCHEDULE_READ;
        }
        else if (appData.receiveDataBuffer[0]==0x03)
        {
            BSP_LED_3Toggle();
            appData.stateUSB = USB_STATE_SCHEDULE_READ;
        }
        else if (appData.receiveDataBuffer[0]==0x00)
        {
            appData.stateUSB = USB_STATE_SEND_REPORT;
        }
        else
        {
            appData.stateUSB = USB_STATE_SCHEDULE_READ;
        }
    }

    break;

case USB_STATE_SEND_REPORT:

    if(appData.hidDataTransmitted)
    {
        if( BSP_SwitchStateGet(BSP_SWITCH_1) == BSP_SWITCH_STATE_PRESSED )
        {

```

```

        appData.transmitDataBuffer[0] = 0x00;
    }
    else
    {
        appData.transmitDataBuffer[0] = 0x01;
    }

    appData.hidDataTransmitted = false;
    USB_DEVICE_HID_ReportSend (USB_DEVICE_HID_INDEX_0,
        &appData.txTransferHandle, appData.transmitDataBuffer, 1);
    appData.stateUSB = USB_STATE_SCHEDULE_READ;
}

break;
}
}
else
{

    /* Reset the USB Task State Machine */
    appData.stateUSB = USB_STATE_INIT;

}
}

```

STEP 11: Schedule a new report receive request

To receive a report from the USB host, you need to schedule a report receive request by using the API provided for the USB HID function driver.

Before scheduling the request, the reception flag needs to be set to false to check when the request is completed (it was set to true in the Step 9).

After scheduling the request, the USB State Machine state needs to be moved to the "waiting for data" state.

Inside the "schedule read" state of the switch statement of the `USB_Task()` function, add the code to set the reception flag to false, then schedule a new report receive request. Finally, set the USB State Machine state member of `appData` structure to the state that waits for data from the USB host (`USB_STATE_WAITING_FOR_DATA`):

```

case USB_STATE_SCHEDULE_READ:

    appData.hidDataReceived = false;
    USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
        &appData.rxTransferHandle, appData.receiveDataBuffer, 64);
    appData.stateUSB = USB_STATE_WAITING_FOR_DATA;

    break;

```

STEP 12: Receive, prepare and send reports

When the report is received, the reception flag is set to true; that means there is valid data in the reception buffer. Inside the switch of the `USB_Task()` function, the state is set to `USB_STATE_WAITING_FOR_DATA` and are checked the next commands that are sent by the Linux USB host driver:

- **0x01:** Toggle the LED1. The state is set to `USB_STATE_SCHEDULE_READ`.
- **0x02:** Toggle the LED2. The state is set to `USB_STATE_SCHEDULE_READ`.
- **0x03:** Toggle the LED3. The state is set to `USB_STATE_SCHEDULE_READ`.
- **0x00:** The USB device gets the Pushbutton state. The state is set to `USB_STATE_SEND_REPORT`. The HID device replies with a report to the host, where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed).

```
case USB_STATE_WAITING_FOR_DATA:
```

```
if( appData.hidDataReceived )
{
    if (appData.receiveDataBuffer[0]==0x01)
    {
        BSP_LED_1Toggle();
        appData.stateUSB = USB_STATE_SCHEDULE_READ;
    }
    else if (appData.receiveDataBuffer[0]==0x02)
    {
        BSP_LED_2Toggle();
        appData.stateUSB = USB_STATE_SCHEDULE_READ;
    }
    else if (appData.receiveDataBuffer[0]==0x03)
    {
        BSP_LED_3Toggle();
        appData.stateUSB = USB_STATE_SCHEDULE_READ;
    }
    else if (appData.receiveDataBuffer[0]==0x00)
    {
        appData.stateUSB = USB_STATE_SEND_REPORT;
    }
    else
    {
        appData.stateUSB = USB_STATE_SCHEDULE_READ;
    }
}
```

```
break;
```

```
case USB_STATE_SEND_REPORT:
```

```
if(appData.hidDataTransmitted)
{
```

```

    if( BSP_SwitchStateGet(BSP_SWITCH_1) == BSP_SWITCH_STATE_PRESSED)
    {
        appData.transmitDataBuffer[0] = 0x00;
    }
    else
    {
        appData.transmitDataBuffer[0] = 0x01;
    }

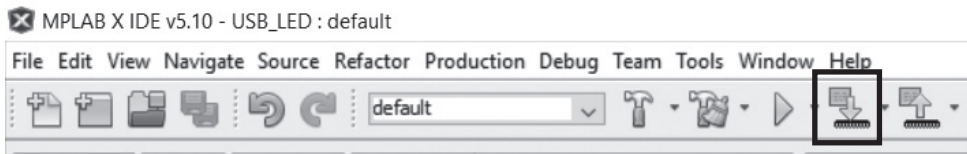
    appData.hidDataTransmitted = false;
    USB_DEVICE_HID_ReportSend (USB_DEVICE_HID_INDEX_0,
        &appData.txTransferHandle, appData.transmitDataBuffer, 1);
    appData.stateUSB = USB_STATE_SCHEDULE_READ;
}

```

STEP 13: Program the application

Power the PIC32MX470 Curiosity Development Board from a Host PC through a Type-A male to mini-B USB cable connected to Mini-B port (J3). Ensure that a jumper is placed in J8 header (between 4 & 3) to select supply from debug USB connector.

Build the code and program the device by clicking on the program button as shown below:



LAB 13.2: "USB LED" module

In the previous lab, you developed the firmware for a fully functional USB HID device that is able to send and receive data by using HID reports. Now, you are going to develop a Linux USB driver to control that USB device. The driver will send USB commands to toggle LED1, LED2 and LED3 of the PIC32MX470 Curiosity Development Board. The driver will receive the command from user space via a sysfs entry, then retransmit it to the PIC32MX HID device. The command values can be 0x01, 0x02 or 0x03. The HID device must toggle LED1 when it receives 0x01, LED2 when it receives 0x02 and LED3 when it receives 0x03 in the report.

You have to stop the hid-generic driver from getting control of our custom driver, so you will include our driver's `USB_VENDOR_ID` and `USB_DEVICE_ID` in the list `hid_ignore_list[]`. Open the `hid-quirks.c` file under the `drivers/hid` folder in the kernel source tree, and add the next line of code (in bold) to the end of the list:


```
static const struct hid_device_id hid_ignore_list[] = {
...
#if IS_ENABLED(CONFIG_MOUSE_SYNAPTICS_USB)
{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_TP) },
{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_INT_TP) },
{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_CPAD) },
{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_STICK) },
{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_WP) },
{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_COMP_TP) },
{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_WTP) },
{ HID_USB_DEVICE(USB_VENDOR_ID_SYNAPTICS, USB_DEVICE_ID_SYNAPTICS_DPAD) },
#endif
{ HID_USB_DEVICE(USB_VENDOR_ID_YEALINK, USB_DEVICE_ID_YEALINK_P1K_P4K_B2K) },
{ HID_USB_DEVICE(0x04d8, 0x003f) },
{ }
};
```

Build the modified kernel, and load it to the target processor:

```
~/linux_rpi3/linux$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- zImage
~/linux_rpi3/linux$ scp arch/arm/boot/zImage root@10.0.0.10:/boot/kernel7.img
```

LAB 13.2 code description of the "USB LED" module

The main code sections of the driver will now be described:

1. Include the function headers:

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

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

```
#define USBLED_VENDOR_ID      0x04D8
#define USBLED_PRODUCT_ID    0x003F

/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
    { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
    { }
};

MODULE_DEVICE_TABLE(usb, id_table);
```

3. Create a private structure that will store the driver's data:

```
struct usb_led {
    struct usb_device *udev;
    u8 led_number;
};
```

4. See below an extract of the probe() routine with the main lines of code to set up the driver commented:

```

static int led_probe(struct usb_interface *interface,
                    const struct usb_device_id *id)
{
    /* Get the usb_device structure from the usb_interface one */
    struct usb_device *udev = interface_to_usbdev(interface);
    struct usb_led *dev = NULL;
    int retval = -ENOMEM;

    dev_info(&interface->dev, "led_probe() function is called.\n");

    /* Allocate our private data structure */
    dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);

    /* store the usb device in our data structure */
    dev->udev = usb_get_dev(udev);

    /* Attach the USB device data to the USB interface */
    usb_set_intfdata(interface, dev);

    /* create a led sysfs entry to interact with the user space */
    device_create_file(&interface->dev, &dev_attr_led);

    return 0;
}

```

5. Write the `led_store()` function. Every time your user space application writes to the led sysfs entry (`/sys/bus/usb/devices/1-1.3:1.0/led`) under the USB device, the driver's `led_store()` function is called. The `usb_led` structure associated to the USB device is recovered by using the `usb_get_intfdata()` function. The command written in the led sysfs entry is stored in the `val` variable. Finally, you will send the command via USB using the `usb_bulk_msg()` function.

The kernel provides the `usb_bulk_msg()` and `usb_control_msg()` helper functions, which make it possible to transfer simple bulk and control messages without having to create an urb structure, initialize it, submit it and wait for its completion handler. These functions are synchronous and will make your code sleep. You must not call them from interrupt context or with a spinlock held.

```

int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data,
                int len, int *actual_length, int timeout);

```

See below a short description of the `usb_bulk_msg()` parameters:

- **usb_dev:** Pointer to the usb device to send the message to.
- **pipe:** Endpoint "pipe" to send the message to.
- **data:** Pointer to the data to send.
- **len:** Length in bytes of the data to send.

- **actual_length:** Pointer to a location to put the actual length transferred in bytes.
- **timeout:** Time in msecs to wait for the message to complete before timing out.

See below an extract of the `led_store()` routine:

```
static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                        const char *buf, size_t count)
{
    struct usb_interface *intf = to_usb_interface(dev);
    struct usb_led *led = usb_get_intfdata(intf);
    u8 val;

    /* transform char array to u8 value */
    kstrtou8(buf, 10, &val);

    led->led_number = val;

    /* Toggle led */
    usb_bulk_msg(led->udev, usb_sndctrlpipe(led->udev, 1),
                  &led->led_number,
                  1,
                  NULL,
                  0);

    return count;
}
static DEVICE_ATTR_RW(led);
```

6. Add a `usb_driver` structure that will be registered to the USB core:

```
static struct usb_driver led_driver = {
    .name = "usbled",
    .probe = led_probe,
    .disconnect = led_disconnect,
    .id_table = id_table,
};
```

7. Register your driver with the USB bus:

```
module_usb_driver(led_driver);
```

8. Create a new `linux_5.4_USB_drivers` folder inside the `linux_5.4_rpi3_drivers` folder:

```
~/linux_5.4_rpi3_drivers$ mkdir linux_5.4_USB_drivers
```

9. Create a new `usb_led.c` file and a Makefile file in the `linux_5.4_USB_drivers` folder, and add `usb_led.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_USB_drivers$ make
~/linux_5.4_rpi3_drivers/linux_5.4_USB_drivers$ make deploy
```

Listing 13-1: usb_led.c

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

#define USBLED_VENDOR_ID      0x04D8
#define USBLED_PRODUCT_ID    0x003F

/* Table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
    { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
    { }
};
MODULE_DEVICE_TABLE(usb, id_table);

struct usb_led {
    struct usb_device *udev;
    u8 led_number;
};

static ssize_t led_show(struct device *dev, struct device_attribute *attr, char *buf)
{
    struct usb_interface *intf = to_usb_interface(dev);
    struct usb_led *led = usb_get_intfdata(intf);

    return sprintf(buf, "%d\n", led->led_number);
}

static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                        const char *buf, size_t count)
{
    struct usb_interface *intf = to_usb_interface(dev);
    struct usb_led *led = usb_get_intfdata(intf);
    u8 val;
    int error, retval;
    dev_info(&intf->dev, "led_store() function is called.\n");

    /* Transform char array to u8 value */
    error = kstrtou8(buf, 10, &val);
    if (error)
        return error;

    led->led_number = val;

    if (val == 1 || val == 2 || val == 3)
        dev_info(&led->udev->dev, "led = %d\n", led->led_number);
    else {
        dev_info(&led->udev->dev, "unknown led %d\n", led->led_number);
        retval = -EINVAL;
        return retval;
    }
}
```

```

/* Toggle led */
retval = usb_bulk_msg(led->udev, usb_sndctrlpipe(led->udev, 1),
                      &led->led_number,
                      1,
                      NULL,
                      0);

if (retval) {
    retval = -EFAULT;
    return retval;
}
return count;
}
static DEVICE_ATTR_RW(led);

static int led_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(interface);
    struct usb_led *dev = NULL;
    int retval = -ENOMEM;

    dev_info(&interface->dev, "led_probe() function is called.\n");

    dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);
    if (!dev) {
        dev_err(&interface->dev, "out of memory\n");
        retval = -ENOMEM;
        goto error;
    }

    dev->udev = usb_get_dev(udev);

    usb_set_intfdata(interface, dev);

    retval = device_create_file(&interface->dev, &dev_attr_led);
    if (retval)
        goto error_create_file;

    return 0;

error_create_file:
    usb_put_dev(udev);
    usb_set_intfdata(interface, NULL);
error:
    kfree(dev);
    return retval;
}

static void led_disconnect(struct usb_interface *interface)
{
    struct usb_led *dev;

    dev = usb_get_intfdata(interface);

    device_remove_file(&interface->dev, &dev_attr_led);

```

```

usb_set_intfdata(interface, NULL);
usb_put_dev(dev->udev);
kfree(dev);

dev_info(&interface->dev, "USB LED now disconnected\n");
}

static struct usb_driver led_driver = {
    .name = "usbled",
    .probe = led_probe,
    .disconnect = led_disconnect,
    .id_table = id_table,
};

module_usb_driver(led_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a synchronous led usb controlled module");

```

usb_led.ko demonstration

Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to one of the four USB HostType-A connectors of the Raspberry Pi board. Power the Raspberry Pi board to boot the processor. Keep the PIC32MX470 board powered off.

Load the module:

```

root@raspberrypi:/home# insmod usb_led.ko
usbcore: registered new interface driver usbled

```

Power now the PIC32MX Curiosity board:

```

root@raspberrypi:/home# usb 1-1.3: new full-speed USB device number 5 using dwc_otg
usb 1-1.3: New USB device found, idVendor=04d8, idProduct=003f, bcdDevice= 1.00
usb 1-1.3: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-1.3: Product: LED_USB HID Demo
usb 1-1.3: Manufacturer: Microchip Technology Inc.
usbled 1-1.3:1.0: led_probe() function is called.

```

Check the new created USB device:

```

root@raspberrypi:/home# cd /sys/bus/usb/devices/1-1.3:1.0
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# ls
authorized          bInterfaceProtocol  ep_01              power
bAlternateSetting   bInterfaceSubClass   ep_81              subsystem
bInterfaceClass      bNumEndpoints        led                supports_autosuspend
bInterfaceNumber     driver                modalias           uevent

```

Read the configurations of the USB device:

```

root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat bNumEndpoints
02
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0/ep_01# cat direction
out

```

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0/ep_81# cat direction
in
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat bAlternateSetting
0
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat bInterfaceClass
03
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat bNumEndpoints
02
```

Switch on the LED1 of the PIC32MX Curiosity board:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 1 > led
usbld 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 1
```

Switch on the LED2 of the PIC32MX Curiosity board:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 2 > led
usbld 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 2
```

Switch on the LED3 of the PIC32MX Curiosity board:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 3 > led
usbld 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 3
```

Read the led status:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# cat led
3
```

Remove the module:

```
root@raspberrypi:/home# rmmod usb_led.ko
usbcore: deregistering interface driver usbld
usbld 1-1.3:1.0: USB LED now disconnected
```

LAB 13.3: "USB LED and Switch" module

In this new lab, you will increase the functionality of the previous driver. Besides controlling three LEDs connected to the USB device, the Linux USB driver will receive a Pushbutton (S1 switch of the PIC32MX470 Curiosity Development Board) state from the USB HID device. The driver will send a command to the USB device with value 0x00, then the HID device will reply with a report where the first byte is the status of the S1 button ("0x00" pressed, "0x01" not pressed). In this driver, unlike the previous one, the communication between the host and the device is done asynchronously by using USB Request Blocks (urbs).

LAB 13.3 code description of the "USB LED and Switch" module

The main code sections of the driver will now be described:

1. Include the function headers:

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

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

```
#define USBLED_VENDOR_ID      0x04D8
#define USBLED_PRODUCT_ID    0x003F

/* Table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
    { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
    { }
};
MODULE_DEVICE_TABLE(usb, id_table);
```

3. Create a private structure that will store the driver's data:

```
struct usb_led {
    struct usb_device      *udev;
    struct usb_interface  *intf;
    struct urb             *interrupt_out_urb;
    struct urb             *interrupt_in_urb;
    struct usb_endpoint_descriptor *interrupt_out_endpoint;
    struct usb_endpoint_descriptor *interrupt_in_endpoint;
    u8                     irq_data;
    u8                     led_number;
    u8                     ibuffer;
    int                    interrupt_out_interval;
    int                    ep_in;
    int                    ep_out;
};
```

4. See below the code of the probe() routine with the main lines of code to configure the driver commented:

```
static int led_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(intf);

    /* Get the current altsetting of the USB interface */
    struct usb_host_interface *altsetting = intf->cur_altsetting;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_led *dev = NULL;
    int ep;
    int ep_in, ep_out;
    int size;
```



```

/*
 * Find the last interrupt out endpoint descriptor
 * to check its number and its size.
 * Just for teaching purposes
 */
usb_find_last_int_out_endpoint(altsetting, &endpoint);

/* Get the endpoint's number */
ep = usb_endpoint_num(endpoint); /* value from 0 to 15, it is 1 */
size = usb_endpoint_maxp(endpoint);
/* Validate endpoint and size */
if (size <= 0) {
    dev_info(&intf->dev, "invalid size (%d)", size);
    return -ENODEV;
}

dev_info(&intf->dev, "endpoint size is (%d)", size);
dev_info(&intf->dev, "endpoint number is (%d)", ep);

/* Get the two addresses (IN and OUT) of the Endpoint 1 */
ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
ep_out = altsetting->endpoint[1].desc.bEndpointAddress;

/* Allocate our private data structure */
dev = kzalloc(sizeof(struct usb_led), GFP_KERNEL);

/* Store values in the data structure */
dev->ep_in = ep_in;
dev->ep_out = ep_out;
dev->udev = usb_get_dev(udev);
dev->intf = intf;

/* Allocate the int_out_urb structure */
dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);

/* Initialize the int_out_urb */
usb_fill_int_urb(dev->interrupt_out_urb,
    dev->udev,
    usb_sndintpipe(dev->udev, ep_out),
    (void *)&dev->irq_data,
    1,
    led_urb_out_callback, dev, 1);

/* Allocate the int_in_urb structure */
dev->interrupt_in_urb = usb_alloc_urb(0, GFP_KERNEL);
if (!dev->interrupt_in_urb)
    goto error_out;

/* Initialize the int_in_urb */
usb_fill_int_urb(dev->interrupt_in_urb,
    dev->udev,
    usb_rcvintpipe(dev->udev, ep_in),
    (void *)&dev->ibuffer,

```

```

        1,
        led_urb_in_callback, dev, 1);

/* Attach the device data to the interface */
usb_set_intfdata(intf, dev);

/* Create the led sysfs entry to interact with the user space */
device_create_file(&intf->dev, &dev_attr_led);

/* Submit the interrupt IN URB */
usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);

return 0;
}

```

5. Write the `led_store()` function. Every time your user space application writes to the led sysfs entry (`/sys/bus/usb/devices/1-1.3:1.0/led`) under the USB device, the driver's `led_store()` function is called. The `usb_led` structure associated to the USB device is recovered by using the `usb_get_intfdata()` function. The command written in the led sysfs entry is stored in the `irq_data` variable. Finally, you will send the command value via USB by using the `usb_submit_urb()` function.

See below an extract of the `led_store()` routine:

```

static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                        const char *buf, size_t count)
{
    struct usb_interface *intf = to_usb_interface(dev);
    struct usb_led *led = usb_get_intfdata(intf);
    u8 val;

    /* transform char array to u8 value */
    kstrtou8(buf, 10, &val);

    led->irq_data = val;

    /* send the data out */
    retval = usb_submit_urb(led->interrupt_out_urb, GFP_KERNEL);

    return count;
}
static DEVICE_ATTR_RW(led);

```

6. Create OUT and IN URB's completion callbacks. The interrupt OUT completion callback merely checks the URB status and returns. The interrupt IN completion callback checks the URB status, then reads the `ibuffer` to know the status received from the PIC32MX board's S1 switch, and finally, re-submits the interrupt IN URB.

```

static void led_urb_out_callback(struct urb *urb)
{
    struct usb_led *dev;

```

```

dev = urb->context;
/* sync/async unlink faults aren't errors */
if (urb->status) {
    if (!(urb->status == -ENOENT ||
        urb->status == -ECONNRESET ||
        urb->status == -ESHUTDOWN))
        dev_err(&dev->udev->dev,
            "%s - nonzero write status received: %d\n",
            __func__, urb->status);
}
}

static void led_urb_in_callback(struct urb *urb)
{
    int retval;
    struct usb_led *dev;

    dev = urb->context;

    if (urb->status) {
        if (!(urb->status == -ENOENT ||
            urb->status == -ECONNRESET ||
            urb->status == -ESHUTDOWN))
            dev_err(&dev->udev->dev,
                "%s - nonzero write status received: %d\n",
                __func__, urb->status);
    }

    if (dev->ibuffer == 0x00)
        pr_info ("switch is ON.\n");
    else if (dev->ibuffer == 0x01)
        pr_info ("switch is OFF.\n");
    else
        pr_info ("bad value received\n");

    usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
}

```

7. Add a `usb_driver` structure that will be registered to the USB core:

```

static struct usb_driver led_driver = {
    .name = "usbled",
    .probe = led_probe,
    .disconnect = led_disconnect,
    .id_table = id_table,
};

```

8. Register your driver with the USB bus:

```
module_usb_driver(led_driver);
```

9. Create a new `usb_urb_int_led.c` file in the `linux_5.4_USB_drivers` folder, and add `usb_urb_int_led.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_USB_drivers$ make
~/linux_5.4_rpi3_drivers/linux_5.4_USB_drivers$ make deploy
```

Listing 13-2: usb_urb_int_led.c

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

#define USBLED_VENDOR_ID    0x04D8
#define USBLED_PRODUCT_ID  0x003F

static void led_urb_out_callback(struct urb *urb);
static void led_urb_in_callback(struct urb *urb);

/* Table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
    { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
    { }
};
MODULE_DEVICE_TABLE(usb, id_table);

struct usb_led {
    struct usb_device *udev;
    struct usb_interface *intf;
    struct urb *interrupt_out_urb;
    struct urb *interrupt_in_urb;
    struct usb_endpoint_descriptor *interrupt_out_endpoint;
    struct usb_endpoint_descriptor *interrupt_in_endpoint;
    u8 irq_data;
    u8 led_number;
    u8 ibuffer;
    int interrupt_out_interval;
    int ep_in;
    int ep_out;
};

static ssize_t led_show(struct device *dev, struct device_attribute *attr, char *buf)
{
    struct usb_interface *intf = to_usb_interface(dev);
    struct usb_led *led = usb_get_intfdata(intf);

    return sprintf(buf, "%d\n", led->led_number);
}

static ssize_t led_store(struct device *dev, struct device_attribute *attr,
                        const char *buf, size_t count)
{
    struct usb_interface *intf = to_usb_interface(dev);
    struct usb_led *led = usb_get_intfdata(intf);
    u8 val;
    int error, retval;
```

```

    dev_info(&intf->dev, "led_store() function is called.\n");

/* Transform char array to u8 value */
error = kstrtou8(buf, 10, &val);
if (error)
    return error;

led->led_number = val;
led->irq_data = val;

if (val == 0)
    dev_info(&led->udev->dev, "read status\n");
else if (val == 1 || val == 2 || val == 3)
    dev_info(&led->udev->dev, "led = %d\n", led->led_number);
else {
    dev_info(&led->udev->dev, "unknown value %d\n", val);
    retval = -EINVAL;
    return retval;
}

/* Send the data out */
retval = usb_submit_urb(led->interrupt_out_urb, GFP_KERNEL);
if (retval) {
    dev_err(&led->udev->dev, "Couldn't submit interrupt_out_urb %d\n", retval);
    return retval;
}

return count;
}
static DEVICE_ATTR_RW(led);

static void led_urb_out_callback(struct urb *urb)
{
    struct usb_led *dev;

    dev = urb->context;

    dev_info(&dev->udev->dev, "led_urb_out_callback() function is called.\n");

/* sync/async unlink faults aren't errors */
if (urb->status) {
    if (!(urb->status == -ENOENT ||
        urb->status == -ECONNRESET ||
        urb->status == -ESHUTDOWN))
        dev_err(&dev->udev->dev,
            "%s - nonzero write status received: %d\n",
            __func__, urb->status);
}
}

static void led_urb_in_callback(struct urb *urb)
{
    int retval;
    struct usb_led *dev;

```

```

dev = urb->context;

dev_info(&dev->udev->dev, "led_urb_in_callback() function is called.\n");

if (urb->status) {
    if (!(urb->status == -ENOENT ||
        urb->status == -ECONNRESET ||
        urb->status == -ESHUTDOWN))
        dev_err(&dev->udev->dev,
            "%s - nonzero write status received: %d\n",
            __func__, urb->status);
}
if (dev->ibuffer == 0x00)
    pr_info ("switch is ON.\n");
else if (dev->ibuffer == 0x01)
    pr_info ("switch is OFF.\n");
else
    pr_info ("bad value received\n");

retval = usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
if (retval)
    dev_err(&dev->udev->dev,
        "Couldn't submit interrupt_in_urb %d\n", retval);
}

static int led_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(intf);
    struct usb_host_interface *altsetting = intf->cur_altsetting;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_led *dev = NULL;
    int ep;
    int ep_in, ep_out;
    int retval, size, res;

    dev_info(&intf->dev, "led_probe() function is called.\n");

    res = usb_find_last_int_out_endpoint(altsetting, &endpoint);
    if (res) {
        dev_info(&intf->dev, "no endpoint found");
        return res;
    }

    ep = usb_endpoint_num(endpoint); /* value from 0 to 15, it is 1 */
    size = usb_endpoint_maxp(endpoint);

    /* Validate endpoint and size */
    if (size <= 0) {
        dev_info(&intf->dev, "invalid size (%d)", size);
        return -ENODEV;
    }

    dev_info(&intf->dev, "endpoint size is (%d)", size);

```

```

dev_info(&intf->dev, "endpoint number is (%d)", ep);

ep_in = altsetting->endpoint[0].desc.bEndpointAddress;
ep_out = altsetting->endpoint[1].desc.bEndpointAddress;

dev_info(&intf->dev, "endpoint in address is (%d)", ep_in);
dev_info(&intf->dev, "endpoint out address is (%d)", ep_out);

dev = kzalloc(sizeof(struct usb_ldev), GFP_KERNEL);

if (!dev)
    return -ENOMEM;

dev->ep_in = ep_in;
dev->ep_out = ep_out;

dev->udev = usb_get_dev(udev);

dev->intf = intf;

/* Allocate int_out_urb structure */
dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);
if (!dev->interrupt_out_urb)
    goto error_out;

/* Initialize int_out_urb */
usb_fill_int_urb(dev->interrupt_out_urb,
                dev->udev,
                usb_sndintpipe(dev->udev, ep_out),
                (void *)&dev->irq_data,
                1,
                led_urb_out_callback, dev, 1);

/* Allocate int_in_urb structure */
dev->interrupt_in_urb = usb_alloc_urb(0, GFP_KERNEL);
if (!dev->interrupt_in_urb)
    goto error_out;

/* Initialize int_in_urb */
usb_fill_int_urb(dev->interrupt_in_urb,
                dev->udev,
                usb_rcvintpipe(dev->udev, ep_in),
                (void *)&dev->ibuffer,
                1,
                led_urb_in_callback, dev, 1);

usb_set_intfdata(intf, dev);

retval = device_create_file(&intf->dev, &dev_attr_led);
if (retval)
    goto error_create_file;

retval = usb_submit_urb(dev->interrupt_in_urb, GFP_KERNEL);
if (retval) {

```

```

        dev_err(&dev->udev->dev,
                "Couldn't submit interrupt_in_urb %d\n", retval);
        device_remove_file(&intf->dev, &dev_attr_led);
        goto error_create_file;
    }

    dev_info(&dev->udev->dev, "int_in_urb submitted\n");

    return 0;

error_create_file:
    usb_free_urb(dev->interrupt_out_urb);
    usb_free_urb(dev->interrupt_in_urb);
    usb_put_dev(udev);
    usb_set_intfdata(intf, NULL);

error_out:
    kfree(dev);
    return retval;
}

static void led_disconnect(struct usb_interface *interface)
{
    struct usb_led *dev;

    dev = usb_get_intfdata(interface);

    device_remove_file(&interface->dev, &dev_attr_led);
    usb_free_urb(dev->interrupt_out_urb);
    usb_free_urb(dev->interrupt_in_urb);
    usb_set_intfdata(interface, NULL);
    usb_put_dev(dev->udev);
    kfree(dev);

    dev_info(&interface->dev, "USB LED now disconnected\n");
}

static struct usb_driver led_driver = {
    .name = "usbled",
    .probe = led_probe,
    .disconnect = led_disconnect,
    .id_table = id_table,
};

module_usb_driver(led_driver);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a led/switch usb controlled module with irq in/out endpoints");

```


usb_urb_int_led.ko demonstration

Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to one of the four USB HostType-A connectors of the Raspberry Pi board. Power the Raspberry Pi board to boot the processor. Keep the PIC32MX470 board powered off.

Load the module:

```
root@raspberrypi:/home# insmod usb_urb_int_led.ko
usb_urb_int_led: loading out-of-tree module taints kernel.
usbcore: registered new interface driver usbled
```

Power now the PIC32MX Curiosity board:

```
root@raspberrypi:/home# usb 1-1.3: new full-speed USB device number 4 using dwc_otg
usb 1-1.3: New USB device found, idVendor=04d8, idProduct=003f, bcdDevice= 1.00
usb 1-1.3: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-1.3: Product: LED_USB HID Demo
usb 1-1.3: Manufacturer: Microchip Technology Inc.
usbled 1-1.3:1.0: led_probe() function is called.
usbled 1-1.3:1.0: endpoint size is (64)
usbled 1-1.3:1.0: endpoint number is (1)
usbled 1-1.3:1.0: endpoint in address is (129)
usbled 1-1.3:1.0: endpoint out address is (1)
usb 1-1.3: int_in_urb submitted
```

Go to the new created USB device:

```
root@raspberrypi:/home# cd /sys/bus/usb/devices/1-1.3:1.0
```

Switch on the LED1 of the PIC32MX Curiosity board:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 1 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 1
usb 1-1.3: led_urb_out_callback() function is called.
```

Switch on the LED2 of the PIC32MX Curiosity board:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 2 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 2
usb 1-1.3: led_urb_out_callback() function is called.
```

Switch on the LED3 of the PIC32MX Curiosity board:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 3 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: led = 3
usb 1-1.3: led_urb_out_callback() function is called.
```

Press the S1 switch of the PIC32MX Curiosity board and get the switch status:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 0 > led
usbled 1-1.3:1.0: led_store() function is called.
usb 1-1.3: read status
usb 1-1.3: led_urb_out_callback() function is called.
usb 1-1.3: led_urb_in_callback() function is called.
```

switch is ON.

Release the S1 switch of PIC32MX Curiosity board and get the switch status:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# echo 0 > led
usbld 1-1.3:1.0: led_store() function is called.
usb 1-1.3: read status
usb 1-1.3: led_urb_out_callback() function is called.
usb 1-1.3: led_urb_in_callback() function is called.
switch is OFF.
```

Remove the module:

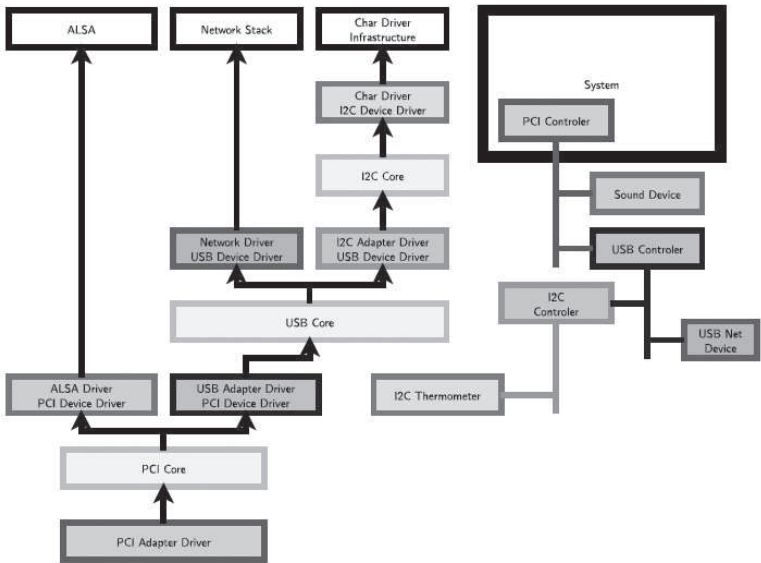
```
root@raspberrypi:/home# rmmod usb_urb_int_led.ko
usbcore: deregistering interface driver usbld
usb 1-1.3: led_urb_in_callback() function is called.
switch is OFF.
usb 1-1.3: Couldn't submit interrupt_in_urb -1
usbld 1-1.3:1.0: USB LED now disconnected
```

LAB 13.4: "I2C to USB Multidisplay LED" module

In the LAB 6.2 of this book, you implemented an I2C Linux driver to control the Analog Devices LTC3206 I2C Multidisplay LED controller (<https://www.analog.com/en/products/ltc3206.html>). In this LAB 13.4, you will write a Linux USB driver to control the LTC3206 from user space by using the I2C Tools for Linux; to perform this task, you will have to create a new I2C adapter within your Linux USB driver.

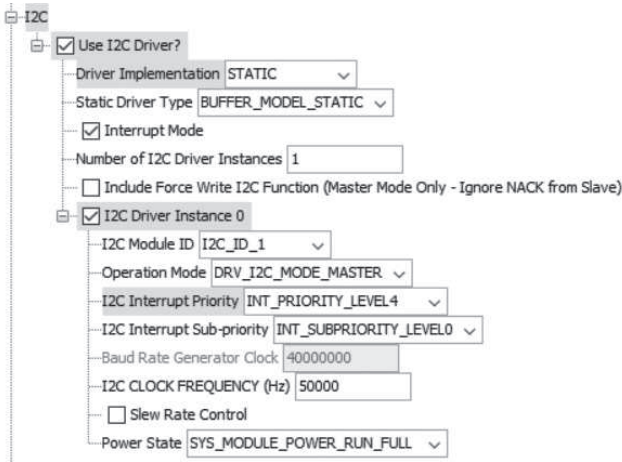
The driver model is recursive. In the following image, you can see all the needed drivers to control an I2C device through a PCI board that integrates a USB to I2C converter. These are the main steps to create this recursive driver model:

- First, you have to develop a PCI device driver that will create a USB adapter (the PCI device driver is the parent of the USB adapter driver).
- Second, you have to develop a USB device driver that will send USB data to the USB adapter driver through the USB core. This USB device driver will also create an I2C adapter driver (the USB device driver is the parent of the I2C adapter driver).
- Finally, you will create an I2C device driver that will send data to the I2C adapter driver through the I2C core and will create a `file_operations` structure to define the driver functions that are called when the Linux user space reads and writes to character devices.



This recursive model will be simplified in the driver of this LAB 13.4, where you are only going to execute the second step of the three previously mentioned. In this driver, the communication between the host and the device is done asynchronously by using an interrupt OUT URB.

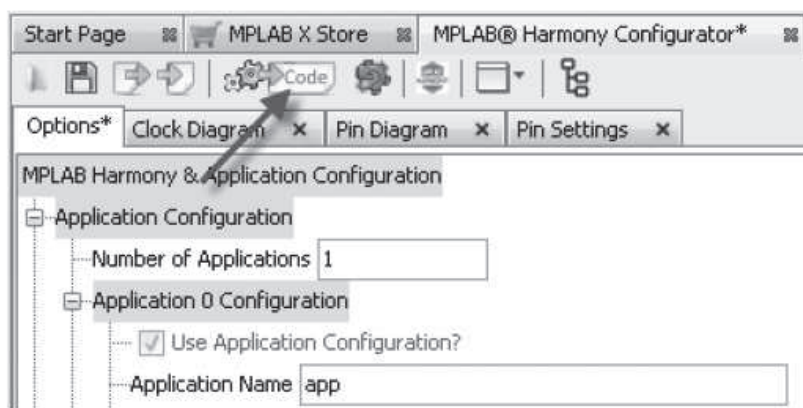
Before developing the Linux driver, you must first add new Harmony configurations to the previous project ones. You must select the I2C Drivers option inside the Harmony Framework Configuration:



In the Pin Table of the MPLAB Harmony Configurator, activate the SCL1 and SDA1 pins of the I2C1 controller:

Output - USB_I2C (Clean, Build, ...)		MPLAB® Harmony Configurator*																							
Output: Pin Table x																									
Package: QFN		RB10	RB11	VSS	VDD	RB12	RB13	RB14	RB15	RF4	RF5	RF3	VBUS	VUSBV...	D-	D+	VDD	RC12	RC15	VSS	RD8	SDA1	SCL1		
Module	Function	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44		
	PGEC3																								
External Interrupt 0	INT0																								
External Interrupt 1	INT1																								
External Interrupt 2	INT2																								
External Interrupt 3	INT3																								
External Interrupt 4	INT4																								
I2C 1 (I2C_ID_1)	SCL1																								
	SDA1																								

Generate the code, save the modified configuration, and generate the project:



Now, you have to modify the generated `app.c` code. Go to the `USB_STATE_WAITING_FOR_DATA` case inside the `USB_Task()` function. Basically, it is waiting for I2C data which has been encapsulated inside a USB interrupt OUT URB. Once the PIC32MX USB device receives the information, it forwards it via I2C to the LTC3206 device connected to the MikroBus 1 of the PIC32MX470 Curiosity Development Board.

```
static void USB_Task (void)
{
    if(appData.usbDeviceIsConfigured)
    {
```

```

switch (appData.stateUSB)
{
    case USB_STATE_INIT:

        appData.hidDataTransmitted = true;
        appData.txTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
        appData.rxTransferHandle = USB_DEVICE_HID_TRANSFER_HANDLE_INVALID;
        appData.stateUSB = USB_STATE_SCHEDULE_READ;

        break;

    case USB_STATE_SCHEDULE_READ:

        appData.hidDataReceived = false;

        /* receive from Host (OUT endpoint). It is a write
           command to the LTC3206 device */
        USB_DEVICE_HID_ReportReceive (USB_DEVICE_HID_INDEX_0,
            &appData.rxTransferHandle, appData.receiveDataBuffer, 64);
        appData.stateUSB = USB_STATE_WAITING_FOR_DATA;

        break;

    case USB_STATE_WAITING_FOR_DATA:

        if( appData.hidDataReceived )
        {

            DRV_I2C_Transmit (appData.drvI2CHandle_Master,
                            0x36,
                            &appData.receiveDataBuffer[0],
                            3,
                            NULL);

            appData.stateUSB = USB_STATE_SCHEDULE_READ;
        }
        break;
    }
}
else
{

    appData.stateUSB = USB_STATE_INIT;

}
}

```

You also need to open the I2C driver inside the APP_Tasks() function:

```

/* Application's initial state. */
case APP_STATE_INIT:
{
    bool appInitialized = true;

```

```

/* Open the I2C Driver for Slave device */
appData.drvI2CHandle_Master = DRV_I2C_Open(DRV_I2C_INDEX_0, DRV_IO_INTENT_WRITE);

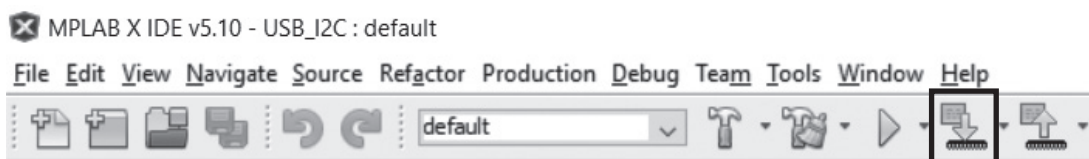
if (appData.drvI2CHandle_Master == (DRV_HANDLE)NULL)
{
    appInitialized = false;
}

/* Open the device layer */
if (appData.handleUsbDevice == USB_DEVICE_HANDLE_INVALID)
{
    appData.handleUsbDevice = USB_DEVICE_Open(USB_DEVICE_INDEX_0,
                                                DRV_IO_INTENT_READWRITE);

    if(appData.handleUsbDevice != USB_DEVICE_HANDLE_INVALID)
    {
        appInitialized = true;
    }
    else
    {
        appInitialized = false;
    }
}
}

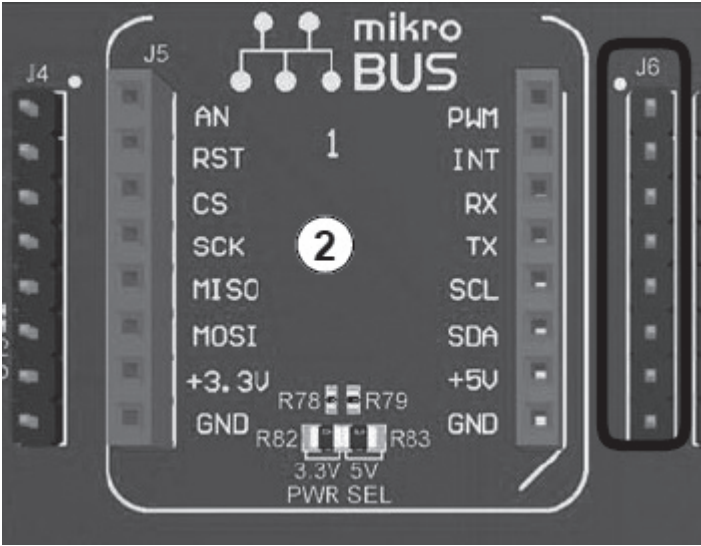
```

Now, you must build the code and program the PIC32MX with the new application. You can download this new project, called USB_I2C, from the GitHub of the book. It is included in the PIC32MX_usb_projects.zip file inside the linux_5.4_USB_drivers folder.



You will use the LTC3206 **DC749A - Demo Board** (<http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc749a.html>) to test the driver. You will connect the board to the MikroBUS 1 connector on the Curiosity PIC32MX470 Development Board. Connect the PIC32MX470 MikroBUS 1 SDA pin to the pin 7 (SDA) of the DC749A J1 connector and the PIC32MX470 MikroBUS 1 SCL pin to the pin 4 (SCL) of the DC749A J1 connector. Connect the PIC32MX470 MikroBUS 1 3.3V pin, the DC749A J20 DVCC pin and the DC749A pin 6 (ENRGB/S) to the DC749A Vin J2 pin. Do not forget to connect GND between the two boards.

Note: For the Curiosity PIC32MX470 Development Board, verify that the value of the series resistors mounted on the SCL and SDA lines of the mikroBUS 1 socket J5 are set to zero Ohms. If not, replace them with zero Ohm resistors. You can also take the SDA and SCL signals from the J6 connector if you do not want to replace the resistors.



LAB 13.4 code description of the "I2C to USB Multidisplay LED" module

The main code sections of the driver will now be described:

1. Include the function headers:

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/usb.h>
#include <linux/i2c.h>
```

2. Create the ID table to support hotplugging. The Vendor ID and Product ID values have to match with the ones used in the PIC32MX USB HID device.

```
#define USBLED_VENDOR_ID      0x04D8
#define USBLED_PRODUCT_ID    0x003F

/* Table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
    { USB_DEVICE(USBLED_VENDOR_ID, USBLED_PRODUCT_ID) },
    { }
};

MODULE_DEVICE_TABLE(usb, id_table);
```

3. Create a private structure that will store the driver data:

```
struct i2c_ltc3206 {
    u8 obuffer[LTC3206_OUTBUF_LEN]; /* USB write buffer */
    /* I2C/SMBus data buffer */
    u8 user_data_buffer[LTC3206_I2C_DATA_LEN]; /* LEN is 3 bytes */
    int ep_out; /* out endpoint */
    struct usb_device *usb_dev; /* the usb device for this device */
    struct usb_interface *interface; /* the interface for this device */
    struct i2c_adapter adapter; /* i2c related things */
    /* wq to wait for an ongoing write */
    wait_queue_head_t usb_urb_completion_wait;
    bool ongoing_usb_ll_op; /* all is in progress */
    struct urb *interrupt_out_urb; /* interrupt out URB */
};
```

4. See below the code of the probe() routine with the main lines of code to configure the driver commented:

```
static int ltc3206_probe(struct usb_interface *interface,
                        const struct usb_device_id *id)
{
    /* Get the current altsetting of the USB interface */
    struct usb_host_interface *hostif = interface->cur_altsetting;
    struct i2c_ltc3206 *dev; /* the data structure */

    /* Allocate data memory for our USB device and initialize it */
    kzalloc(sizeof(*dev), GFP_KERNEL);
```



```

/* Get interrupt ep_out address */
dev->ep_out = hostif->endpoint[1].desc.bEndpointAddress;

dev->usb_dev = usb_get_dev(interface_to_usbdev(interface));
dev->interface = interface;

/* Declare dynamically a wait queue */
init_waitqueue_head(&dev->usb_urb_completion_wait);

/* Save our data pointer in this USB interface device */
usb_set_intfdata(interface, dev);

/* Set up I2C adapter description */
dev->adapter.owner = THIS_MODULE;
dev->adapter.class = I2C_CLASS_HWMON;
dev->adapter.algo = &ltc3206_usb_algorithm;
i2c_set_adapdata(&dev->adapter, dev);

/* Attach the I2C adapter to the USB interface */
dev->adapter.dev.parent = &dev->interface->dev;

/* Initialize the I2C device */
ltc3206_init(dev);

/* Attach the adapter to the I2C layer */
i2c_add_adapter(&dev->adapter);

return 0;
}

```

5. Write the `ltc3206_init()` function. Inside this function, you will allocate and initialize the interrupt OUT URB which is used for the communication between the host and the device. See below the code of the `ltc3206_init()` routine:

```

static int ltc3206_init(struct i2c_ltc3206 *dev)
{
    /* Allocate int_out_urb structure */
    interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);

    /* Initialize int_out_urb structure */
    usb_fill_int_urb(dev->interrupt_out_urb, dev->usb_dev,
                    usb_sndintpipe(dev->usb_dev, dev->ep_out),
                    (void *)&dev->obuffer, LTC3206_OUTBUF_LEN,
                    ltc3206_usb_cmpl_cbk, dev,
                    1);

    return 0;
}

```

6. Create an `i2c_algorithm` structure that represents the I2C transfer method. You will initialize two variables inside this structure:

- **master_xfer**: Issues a set of I2C transactions defined by the `msgs` array, with `num` messages available to transfer using the adapter specified by `adap`.
- **functionality**: Returns the flags that the adapter supports.

```
static const struct i2c_algorithm ltc3206_usb_algorithm = {
    .master_xfer = ltc3206_usb_i2c_xfer,
    .functionality = ltc3206_usb_func,
};
```

7. Write the `ltc3206_usb_i2c_xfer()` function, which will be called each time you write to the I2C adapter from user space. The `ltc3206_usb_i2c_xfer()` function will call `ltc3206_i2c_write()`, which stores the I2C data received from user space in the `obuferr[]` char array, then calls `ltc3206_ll_cmd()`, which submits the interrupt OUT URB to the USB device and waits for the URB's completion.

```
static int ltc3206_usb_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
{
    /* get the private data structure */
    struct i2c_ltc3206 *dev = i2c_get_adapdata(adap);
    struct i2c_msg *pmsg;
    int ret, count;

    pr_info("number of i2c msgs is = %d\n", num);

    for (count = 0; count < num; count++) {
        pmsg = &msgs[count];
        ret = ltc3206_i2c_write(dev, pmsg);
        if (ret < 0)
            goto abort;
    }

    /* if all the messages were transferred ok, return "num" */
    ret = num;

abort:
    return ret;
}

static int ltc3206_i2c_write(struct i2c_ltc3206 *dev, struct i2c_msg *pmsg)
{
    u8 ucXferLen;
    int rv;
    u8 *pSrc, *pDst;

    /* I2C write lenght */
    ucXferLen = (u8)pmsg->len;

    pSrc = &pmsg->buf[0];
    pDst = &dev->obuferr[0];
    memcpy(pDst, pSrc, ucXferLen);
}
```

```

pr_info("oubuffer[0] = %d\n", dev->oubuffer[0]);
pr_info("oubuffer[1] = %d\n", dev->oubuffer[1]);
pr_info("oubuffer[2] = %d\n", dev->oubuffer[2]);

rv = ltc3206_ll_cmd(dev);
if (rv < 0)
    return -EFAULT;

return 0;
}

static int ltc3206_ll_cmd(struct i2c_ltc3206 *dev)
{
    int rv;

    /*
     * tell everybody to leave the URB alone
     * we are going to write to the LTC3206
     */
    dev->ongoing_usb_ll_op = 1; /* doing USB communication */

    /* submit the interrupt out ep packet */
    if (usb_submit_urb(dev->interrupt_out_urb, GFP_KERNEL)) {
        dev_err(&dev->interface->dev,
                "ltc3206(ll): usb_submit_urb intr out failed\n");
        dev->ongoing_usb_ll_op = 0;
        return -EIO;
    }

    /* wait for its completion, the USB URB callback will signal it */
    rv = wait_event_interruptible(dev->usb_urb_completion_wait,
                                (!dev->ongoing_usb_ll_op));
    if (rv < 0) {
        dev_err(&dev->interface->dev, "ltc3206(ll): wait
                                interrupted\n");
        goto ll_exit_clear_flag;
    }

    return 0;

ll_exit_clear_flag:
    dev->ongoing_usb_ll_op = 0;
    return rv;
}

```

8. Create the interrupt OUT URB's completion callback. The completion callback checks the URB status and re-submits the URB if there was an error status. If the transmission was successful, the callback wakes up the sleeping process and returns.

```

static void ltc3206_usb_cmpl_cbk(struct urb *urb)
{
    struct i2c_ltc3206 *dev = urb->context;

```

```

    int status = urb->status;
    int retval;

    switch (status) {
    case 0:                /* success */
        break;
    case -ECONNRESET:      /* unlink */
    case -ENOENT:
    case -ESHUTDOWN:
        return;
    /* -EPIPE: should clear the halt */
    default:               /* error */
        goto resubmit;
    }

    /*
     * wake up the waiting function
     * modify the flag indicating the ll status
     */
    dev->ongoing_usb_ll_op = 0; /* communication is OK */
    wake_up_interruptible(&dev->usb_urb_completion_wait);
    return;

resubmit:
    retval = usb_submit_urb(urb, GFP_ATOMIC);
    if (retval) {
        dev_err(&dev->interface->dev,
                "ltc3206(irq): can't resubmit intrerrupt urb, retval %d\n",
                retval);
    }
}

```

9. Add a `usb_driver` structure that will be registered to the USB core:

```

static struct usb_driver ltc3206_driver = {
    .name = DRIVER_NAME,
    .probe = ltc3206_probe,
    .disconnect = ltc3206_disconnect,
    .id_table = ltc3206_table,
};

```

10. Register your driver with the USB bus:

```
module_usb_driver(ltc3206_driver);
```

11. Create a new `usb_ltc3206.c` file in the `linux_5.4_USB_drivers` folder, and add `usb_ltc3206.o` to your Makefile `obj-m` variable, then build and deploy the module to the Raspberry Pi:

```
~/linux_5.4_rpi3_drivers/linux_5.4_USB_drivers$ make
```

```
~/linux_5.4_rpi3_drivers/linux_5.4_USB_drivers$ make deploy
```

Listing 13-3: usb_ltc3206.c

```

#include <linux/module.h>
#include <linux/slab.h>
#include <linux/usb.h>
#include <linux/i2c.h>

#define DRIVER_NAME "usb-ltc3206"

#define USB_VENDOR_ID_LTC3206      0x04d8
#define USB_DEVICE_ID_LTC3206      0x003f

#define LTC3206_OUTBUF_LEN         3      /* USB write packet length */
#define LTC3206_I2C_DATA_LEN       3

/* Structure to hold all of our device specific stuff */
struct i2c_ltc3206 {
    u8 obuffer[LTC3206_OUTBUF_LEN];      /* USB write buffer */
    /* I2C/SMBus data buffer */
    u8 user_data_buffer[LTC3206_I2C_DATA_LEN];
    int ep_out;                          /* out endpoint */
    struct usb_device *usb_dev;          /* the usb device for this device */
    struct usb_interface *interface;     /* the interface for this device */
    struct i2c_adapter adapter;          /* i2c related things */

    /* wq to wait for an ongoing write */
    wait_queue_head_t usb_urb_completion_wait;
    bool ongoing_usb_ll_op;              /* all is in progress */
    struct urb *interrupt_out_urb;
};

/*
 * Return list of supported functionality.
 */
static u32 ltc3206_usb_func(struct i2c_adapter *a)
{
    return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL | I2C_FUNC_SMBUS_READ_BLOCK_DATA | I2C_FUNC_
SMBUS_BLOCK_PROC_CALL;
}

/* usb out urb callback function */
static void ltc3206_usb_cmpl_cbk(struct urb *urb)
{
    struct i2c_ltc3206 *dev = urb->context;
    int status = urb->status;
    int retval;

    switch (status) {
    case 0:                             /* success */
        break;
    case -ECONNRESET:                   /* unlink */
    case -ENOENT:
    case -ESHUTDOWN:

```

```

        return;
/* -EPIPE: should clear the halt */
default:      /* error */
    goto resubmit;
}

/*
 * Wake up the waiting function.
 * Modify the flag indicating the ll status
 */
dev->ongoing_usb_ll_op = 0; /* communication is OK */
wake_up_interruptible(&dev->usb_urb_completion_wait);
return;

resubmit:
    retval = usb_submit_urb(urb, GFP_ATOMIC);
    if (retval) {
        dev_err(&dev->interface->dev,
                "ltc3206(irq): can't resubmit interrupt urb, retval %d\n",
                retval);
    }
}

static int ltc3206_ll_cmd(struct i2c_ltc3206 *dev)
{
    int rv;

    /*
     * Tell everybody to leave the URB alone.
     * We are going to write to the LTC3206 device
     */
    dev->ongoing_usb_ll_op = 1; /* doing USB communication */

    /* Submit the interrupt out URB packet */
    if (usb_submit_urb(dev->interrupt_out_urb, GFP_KERNEL)) {
        dev_err(&dev->interface->dev,
                "ltc3206(ll): usb_submit_urb intr out failed\n");
        dev->ongoing_usb_ll_op = 0;
        return -EIO;
    }

    /* Wait for the transmit completion. The USB URB callback will signal it */
    rv = wait_event_interruptible(dev->usb_urb_completion_wait, (!dev->ongoing_usb_ll_op));
    if (rv < 0) {
        dev_err(&dev->interface->dev, "ltc3206(ll): wait interrupted\n");
        goto ll_exit_clear_flag;
    }

    return 0;

ll_exit_clear_flag:
    dev->ongoing_usb_ll_op = 0;
    return rv;
}

```

```

static int ltc3206_init(struct i2c_ltc3206 *dev)
{
    int ret;

    /* Initialize the LTC3206 */
    dev_info(&dev->interface->dev,
        "LTC3206 at USB bus %03d address %03d -- ltc3206_init()\n",
        dev->usb_dev->bus->busnum, dev->usb_dev->devnum);

    /* Allocate the int out URB */
    dev->interrupt_out_urb = usb_alloc_urb(0, GFP_KERNEL);
    if (!dev->interrupt_out_urb) {
        ret = -ENODEV;
        goto init_error;
    }

    /* Initialize the int out URB */
    usb_fill_int_urb(dev->interrupt_out_urb, dev->usb_dev,
        usb_sndintpipe(dev->usb_dev, dev->ep_out),
        (void *)&dev->obuffer, LTC3206_OUTBUF_LEN,
        ltc3206_usb_cmpl_cbk, dev,
        1);

    ret = 0;

    goto init_no_error;

init_error:
    dev_err(&dev->interface->dev, "ltc3206_init: Error = %d\n", ret);
    return ret;

init_no_error:
    dev_info(&dev->interface->dev, "ltc3206_init: Success\n");
    return ret;
}

static int ltc3206_i2c_write(struct i2c_ltc3206 *dev, struct i2c_msg *pmsg)
{
    u8 ucXferLen;
    int rv;
    u8 *pSrc, *pDst;

    if (pmsg->len > LTC3206_I2C_DATA_LEN)
    {
        pr_info ("problem with the lenght\n");
        return -EINVAL;
    }

    /* I2C write lenght */
    ucXferLen = (u8)pmsg->len;

    pSrc = &pmsg->buf[0];
    pDst = &dev->obuffer[0];

```

```

memcpy(pDst, pSrc, ucXferLen);

pr_info("oubuffer[0] = %d\n", dev->oubuffer[0]);
pr_info("oubuffer[1] = %d\n", dev->oubuffer[1]);
pr_info("oubuffer[2] = %d\n", dev->oubuffer[2]);

rv = ltc3206_ll_cmd(dev);
if (rv < 0)
    return -EFAULT;

return 0;
}

/* Device layer, called from the I2C user app */
static int ltc3206_usb_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
{
    struct i2c_ltc3206 *dev = i2c_get_adapdata(adap);
    struct i2c_msg *pmsg;
    int ret, count;

    pr_info("number of i2c msgs is = %d\n", num);

    for (count = 0; count < num; count++) {
        pmsg = &msgs[count];
        ret = ltc3206_i2c_write(dev, pmsg);
        if (ret < 0)
            goto abort;
    }

    /* If all the messages were transferred ok, return "num" */
    ret = num;
abort:
    return ret;
}

static const struct i2c_algorithm ltc3206_usb_algorithm = {
    .master_xfer = ltc3206_usb_i2c_xfer,
    .functionality = ltc3206_usb_func,
};

static const struct usb_device_id ltc3206_table[] = {
    { USB_DEVICE(USB_VENDOR_ID_LTC3206, USB_DEVICE_ID_LTC3206) },
    { }
};
MODULE_DEVICE_TABLE(usb, ltc3206_table);

static void ltc3206_free(struct i2c_ltc3206 *dev)
{
    usb_put_dev(dev->usb_dev);
    usb_set_intfdata(dev->interface, NULL);
    kfree(dev);
}

```



```

static int ltc3206_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
    struct usb_host_interface *hostif = interface->cur_altsetting;
    struct i2c_ltc3206 *dev;
    int ret;

    dev_info(&interface->dev, "ltc3206_probe() function is called.\n");

    /* Allocate memory for our device and initialize it */
    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
    if (dev == NULL) {
        pr_info("i2c-ltc3206(probe): no memory for device state\n");
        ret = -ENOMEM;
        goto error;
    }

    /* Get ep_out address */
    dev->ep_out = hostif->endpoint[1].desc.bEndpointAddress;

    dev->usb_dev = usb_get_dev(interface_to_usbdev(interface));
    dev->interface = interface;

    init_waitqueue_head(&dev->usb_urb_completion_wait);

    /* Save our data pointer in this interface device */
    usb_set_intfdata(interface, dev);

    /* Set up I2C adapter description */
    dev->adapter.owner = THIS_MODULE;
    dev->adapter.class = I2C_CLASS_HWMON;
    dev->adapter.algo = &ltc3206_usb_algorithm;
    i2c_set_adapdata(&dev->adapter, dev);

    snprintf(dev->adapter.name, sizeof(dev->adapter.name),
             DRIVER_NAME " at bus %03d device %03d",
             dev->usb_dev->bus->busnum, dev->usb_dev->devnum);

    dev->adapter.dev.parent = &dev->interface->dev;

    /* Initialize the ltc3206 device */
    ret = ltc3206_init(dev);
    if (ret < 0) {
        dev_err(&interface->dev, "failed to initialize adapter\n");
        goto error_init;
    }

    /* Attach to I2C layer */
    ret = i2c_add_adapter(&dev->adapter);
    if (ret < 0) {
        dev_info(&interface->dev, "failed to add I2C adapter\n");
        goto error_i2c;
    }

    dev_info(&dev->interface->dev, "ltc3206_probe() -> chip connected -> Success\n");
}

```

```
    return 0;

error_init:
    usb_free_urb(dev->interrupt_out_urb);

error_i2c:
    usb_set_intfdata(interface, NULL);
    ltc3206_free(dev);
error:
    return ret;
}

static void ltc3206_disconnect(struct usb_interface *interface)
{
    struct i2c_ltc3206 *dev = usb_get_intfdata(interface);

    i2c_del_adapter(&dev->adapter);

    usb_kill_urb(dev->interrupt_out_urb);
    usb_free_urb(dev->interrupt_out_urb);

    usb_set_intfdata(interface, NULL);
    ltc3206_free(dev);

    pr_info("i2c-ltc3206(disconnect) -> chip disconnected");
}

static struct usb_driver ltc3206_driver = {
    .name = DRIVER_NAME,
    .probe = ltc3206_probe,
    .disconnect = ltc3206_disconnect,
    .id_table = ltc3206_table,
};

module_usb_driver(ltc3206_driver);

MODULE_AUTHOR("Alberto Liberal <aliberal@arroweurope.com>");
MODULE_DESCRIPTION("This is a usb controlled i2c ltc3206 device");
MODULE_LICENSE("GPL");
```

usb_ltc3206.ko demonstration

Connect the PIC32MX470 Curiosity Development Board USB Micro-B port (J12) to one of the four USB HostType-A connectors of the Raspberry Pi board. Power the Raspberry Pi board to boot the processor. Keep the PIC32MX470 board powered off.

Check the I2C adapters of the Raspberry Pi board:

```
root@raspberrypi:/home# i2cdetect -l
i2c-1  i2c          bcm2835 (i2c@7e804000)          I2C adapter
```

Load the module:

```
root@raspberrypi:/home# insmod usb_ltc3206.ko
usb_ltc3206: loading out-of-tree module taints kernel.
usbcore: registered new interface driver usb-ltc3206
```

Power now the PIC32MX Curiosity board:

```
root@raspberrypi:/home# usb 1-1.3: new full-speed USB device number 4 using dwc_otg
usb 1-1.3: New USB device found, idVendor=04d8, idProduct=003f, bcdDevice= 1.00
usb 1-1.3: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-1.3: Product: USB to I2C demo
usb 1-1.3: Manufacturer: Microchip Technology Inc.
usb-ltc3206 1-1.3:1.0: ltc3206_probe() function is called.
usb-ltc3206 1-1.3:1.0: LTC3206 at USB bus 001 address 004 -- ltc3206_init()
usb-ltc3206 1-1.3:1.0: ltc3206_init: Success
usb-ltc3206 1-1.3:1.0: ltc3206_probe() -> chip connected -> Success
```

Check again the I2C adapters of the Raspberry board. You will find a new adapter:

```
root@raspberrypi:/home# i2cdetect -l
i2c-1  i2c          bcm2835 (i2c@7e804000)          I2C adapter
i2c-11 i2c          usb-ltc3206 at bus 001 device 004      I2C adapter
```

See the entries under the new USB device:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# ls
authorized          bInterfaceProtocol ep_01      power
bAlternateSetting   bInterfaceSubClass ep_81      subsystem
bInterfaceClass      bNumEndpoints    i2c-4     supports_autosuspend
bInterfaceNumber     driver            modalias   uevent
```

Verify the communication between the USB host and the USB device. The next commands toggle the three leds of the PIC32MX board and set maximum brightness of the LTC3206 blue LED:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x00 0xf0 0x00 i
number of i2c msgs is = 1
oubuffer[0] = 0
oubuffer[1] = 240
oubuffer[2] = 0
```

Set maximum brightness of the LTC3206 red LED:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0xf0 0x00 0x00 i
```

Decrease the brightness of the LTC3206 red LED:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x10 0x00 0x00 i
```

Set maximum brightness of the LTC3206 green LED:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x00 0x0f 0x00 i
```

Set maximum brightness of the LTC3206 green LED and the LTC3206 SUB display:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x00 0x0f 0x0f i
```

Set maximum brightness of the LTC3206 MAIN display:

```
root@raspberrypi:/sys/bus/usb/devices/1-1.3:1.0# i2cset -y 11 0x1b 0x00 0x00 0xf0 i
```

Remove the module:

```
root@raspberrypi:/home# rmmod usb_ltc3206.ko
usbcore: deregistering interface driver usb-ltc3206
```

Power off the PIC32MX Curiosity board:

```
root@raspberrypi:/home# i2c-ltc3206(disconnect) -> chip disconnected
usb 1-1.3: USB disconnect, device number 4
```

References

1. Raspberry Pi Linux documentation.
<https://www.raspberrypi.org/documentation/linux/>
2. Broadcom, "BCM2835 ARM Peripherals guide".
<https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf>
3. Rubini, Corbet, and Kroah-Hartman, "Linux Device Drivers", third edition, O'Reilly, 02/2005.
<https://lwn.net/Kernel/LDD3/>
4. bootlin, "Linux Kernel and Driver Development Training".
<https://bootlin.com/doc/training/linux-kernel/>
5. Corbet, LWN.net, Kroah-Hartman, The Linux Foundation, "Linux Kernel Development report", seventh edition, 08/2016.
<https://www.linuxfoundation.org/events/2016/08/linux-kernel-development-2016/>
6. The Linux kernel Device Tree documentation.
<https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt>
7. The Device Tree specification.
<https://github.com/devicetree-org/devicetree-specification/releases>
8. The Raspberry Pi Device Tree documentation.
<https://www.raspberrypi.org/documentation/configuration/device-tree.md>
9. The Linux kernel General Purpose Input/Output (GPIO) documentation.
<https://www.kernel.org/doc/html/v5.4/driver-api/gpio/index.html>
10. The Linux kernel PINCTRL (PIN CONTROL) subsystem documentation.
<https://www.kernel.org/doc/html/v4.15/driver-api/pinctl.html>
11. Hans-Jürgen Koch, "The Userspace I/O HOWTO".
<http://www.hep.by/gnu/kernel/uio-howto/>
12. The Linux kernel I2C/SMBus subsystem documentation.
<https://www.kernel.org/doc/html/v5.4/i2c/index.html>
13. The Linux kernel IRQ domain documentation.
<https://www.kernel.org/doc/Documentation/IRQ-domain.txt>
14. The Linux kernel generic IRQ documentation.
<https://www.kernel.org/doc/html/latest/core-api/genericirq.html>

15. The Linux kernel Concurrency Managed Workqueue (cmwq) documentation.
<https://www.kernel.org/doc/html/latest/core-api/workqueue.html>
16. The Linux kernel Deferred work labs.
https://linux-kernel-labs.github.io/refs/heads/master/labs/deferred_work.html
17. The Linux kernel DMAEngine documentation.
<https://www.kernel.org/doc/html/latest/driver-api/dmaengine/index.html>
18. The Linux kernel Input documentation.
<https://www.kernel.org/doc/html/latest/input/index.html>
19. The pygame library documentation.
<https://www.pygame.org/docs/>
20. Daniel Baluta, "Industrial I/O driver developer's guide".
<https://dbaluta.github.io/>
21. The libgpod library documentation.
<https://git.kernel.org/pub/scm/libs/libgpod/libgpod.git/about/>
22. Raghavendra Chandra Ganiga, "regmap: Reducing the Redundancy in Linux Code".
<https://www.opensourceforu.com/2017/01/regmap-reducing-redundancy-linux-code/>
23. Silicon Labs, "Human Interface Device Tutorial". Application note number: AN249
<https://www.silabs.com/documents/public/application-notes/AN249.pdf>
24. Exar, "USB Basics for the EXAR Family of USB Uarts". Application note number: AN213
<https://www.exar.com/appnote/an213.pdf>
25. The Linux kernel USB API documentation.
<https://www.kernel.org/doc/html/v5.4/driver-api/usb/index.html>

Index

A

`alloc_chrdev_region()` **function** 72-73

B

binding

matching, device and driver 44, 91, 167, 380, 414

Bootloader 20-21

`bus_register()` **function** 42-43, 163, 413

`bus_type` **structure** 41, 42-45, 163, 413

C

C runtime library

about 25

glibc 25-26

`cdev` **structure** 49, 50

`cdev_add()` **function** 73

`cdev_init()` **function** 73

character device

about 69-70

devtmpfs, creation 79-80

major and minor numbers 72

misc framework, creation 85-86

character device driver

`alloc_chrdev_region()` **function** 72-73

`cdev_add()` **function** 73

`cdev_init()` **function** 73

`copy_from_user()` **function** 113

`copy_to_user()` **function** 113

file_operations **structure** 70

MAKEDEV script 71-72

`register_chrdev_region()` **function** 72-73

`class_create()` **function** 80

`class_destroy()` **function** 80

`device_create()` **function** 80

`device_destroy()` **function** 80

`copy_from_user()` **function** 113

`copy_to_user()` **function** 113

`create_singlethread_workqueue()` **function** 225-226

`create_workqueue()` **function** 225-226

D

`DECLARE_WAIT_QUEUE_HEAD()` **function** 228

`DECLARE_WORK()` **function** 224

deferred work

about 214-215

bottom-half, about 215

softirqs 215-216

tasklets 217

threaded interrupts 221-222

timers 217-218

top-half, about 215

workqueues 223-226

`delayed_work` **structure** 223

`del_timer()` **function** 218

`del_timer_sync()` **function** 218

`destroy_workqueue()` **function** 225

device node 69

Device Tree

about 53

building, on Raspberry Pi 39

chosen node 56

- compatible property, about 53
- location 54
- machine_desc **structure** 54-55
- of_platform_populate() **function** 58, 164, 167, 413
- of_scan_flat_dt() **function** 56-57
- overlay 325-331
- setup_machine_fdt() **function** 56, 57
- unflatten_device_tree() **function** 58
- Device Tree, bindings**
 - about 53
 - clocks 417
 - gpios 112, 120, 169, 478
 - I2C devices 168
 - interrupts 204, 205
 - pin controller 97
 - resource properties 132
 - spi controller 417
 - spi devices 418-419
- device **structure** 46
- device_create() **function** 80
- device_destroy() **function** 80
- device_driver **structure** 41, 45
- device_for_each_child_node() **function** 181
- device_get_child_node_count() **function** 181, 182, 244
- device_register() **function** 43
- devm_gpiochip_add_data() **function** 266, 293, 295
- devm_gpiod_get() **function** 110
- devm_gpiod_get_index() **function** 110
- devm_gpiod_put() **function** 110
- devm_iio_device_alloc() **function** 446
- devm_iio_device_register() **function** 447
- devm_iio_triggered_buffer_setup() **function** 452
- devm_input_allocate_polled_device() **function** 378, 390
- devm_ioremap() **function** 114-115, 132, 136, 139, 154
- devm_iounmap() **function** 114
- devm_kmalloc() **function** 343
- devm_kzalloc() **function** 139, 170, 182, 244, 378
- devm_led_classdev_register() **function** 134, 139, 178, 182
- devm_pinctrl_register() **function** 104, 296
- devm_regmap_init_i2c() **function** 532
- devm_regmap_init_spi() **function** 532
- devm_request_irq() **function** 206-207
- devm_request_threaded_irq() **function** 221-222
- Direct Memory Access (DMA)**
 - about 353
 - cache coherency 353-354
 - dma_map_ops **structure** 354
- DMA engine Linux API**
 - about 355
 - bus address 357, 360
 - Contiguous Memory Allocator (CMA) 357
 - Descriptor, for transaction 355
 - dmaengine_prep_slave_sg() **function** 356
 - dmaengine_slave_config() **function** 355
 - dma_async_issue_pending() **function** 356
 - dma_request_chan() **function** 355
 - dma_slave_config **structure** 355
- DMA mapping, coherent**
 - about 357-359
 - dma_alloc_coherent() **function** 357-359
 - dma_free_coherent() **function** 359
- DMA mapping, streaming**
 - about 359-361
 - dma_map_single() **function** 359, 360
 - dma_unmap_single() **function** 359
 - dmaengine_prep_slave_sg() **function** 356
 - dmaengine_slave_config() **function** 355
 - dmaengine_submit() **function** 356
 - dma_alloc_coherent() **function** 357-359
 - dma_async_issue_pending() **function** 356
 - dma_free_coherent() **function** 359

dma_map_ops structure 354
dma_map_sg() function 356
dma_map_single() function 359, 360
dma_request_chan() function 355
dma_unmap_single() function 359
documentation, for Raspberry Pi 96
driver_register() function 43-44

E

ethernet, setting up 36
exit() function 61

F

file_operations structure 70
flush_scheduled_work() function 224
flush_workqueue() function 225
for_each_child_of_node() function 136, 139, 182, 244
free_irq() function 111, 206
fwnode_handle structure 180, 182, 244

G

get_user() function 113
glibc 25-26
GPIO controller driver
 about 106-109
 gpio_chip structure 107, 478
 gpio_irq_chip structure 203, 293, 294

GPIO interface, descriptor based

devm_gpiod_get() function 110
devm_gpiod_get_index() function 110
devm_gpiod_put() function 110
gpiod_direction_input() function 111, 244
gpiod_direction_output() function 111
gpiod_to_irq() function 111, 200, 209, 244
mapping, GPIOs to IRQs 111

obtaining GPIOs 110
using GPIOs 110-111

GPIO irqchips, categories 108-109, 200-204
gpiod_direction_input() function 111, 244
gpiod_direction_output() function 111
gpiod_to_irq() function 111, 200, 209, 244
gpio_chip.to_irq() function 200

H

hardware irq's (hwirq) 195, 197, 199, 200, 209, 269

I

I2C, definition 161

I2C device driver

i2c_add_driver() function 166
i2c_device_id structure 166-167
i2c_driver structure 165-166
registering 165-166

I2C subsystem, Linux

I2C bus core 163
I2C controller drivers 163-164
I2C device drivers 164

i2cdetect, application 176, 192, 289, 383, 633

i2c_adapter structure 163, 167

i2c_add_driver() function 166

i2c_del_driver() function 166

i2c_set_clientdata() function 170, 182, 378, 390

IIO buffer

about 450
iio_chan_spec structure 450-451
iio_push_to_buffers_with_timestamp() function 452, 453, 544
setup 450-452
sysfs interface 450

IIO device, channels

about 447

- iio_chan_spec structure** 447-448
 - IIO device, sysfs interface** 447
 - IIO driver**
 - devm_iio_device_alloc() function** 446
 - devm_iio_device_register() function** 447
 - IIO events**
 - about** 454
 - iio_event_spec structure** 454-455
 - iio_push_event() function** 456, 537, 543
 - kernel hooks** 456
 - sysfs attributes** 454-455
 - IIO triggered buffers**
 - about** 452-453
 - devm_iio_triggered_buffer_setup() function** 452, 544-545
 - iio_triggered_buffer_cleanup() function** 452
 - iio_triggered_buffer_setup() function** 452
 - IIO utils** 458
 - iio_info structure**
 - about** 449
 - kernel hooks** 449
 - iio_priv() function** 472, 516
 - iio_push_event() function** 456, 537, 543
 - iio_push_to_buffers_with_timestamp() function** 452, 453, 544
 - iio_triggered_buffer_cleanup() function** 452
 - iio_triggered_buffer_setup() function** 452
 - Industrial I/O framework (IIO), about** 445
 - init process** 29
 - init programs** 29
 - init() function** 61
 - init_waitqueue_head() function** 228, 232
 - INIT_WORK() function** 224
 - Input subsystem framework**
 - about** 371
 - drivers** 372-373
 - evtest, application** 373, 374, 383, 398, 402, 443, 529
 - input_event() function** 375
 - input_polled_dev structure** 374-375
 - input_register_polled_device() function** 375, 380, 391
 - input_sync() function** 375
 - input_unregister_polled_device() function** 375
 - set_bit() function** 375
 - interrupt context, kernel** 113, 206, 207, 214, 215, 216, 217, 221, 222, 359, 360, 453, 544, 583
 - interrupt handler**
 - about** 206
 - function, parameters** 206-207
 - function, return values** 207
 - interrupts, Device Tree** 204-205
 - ioremap() function** 114
 - IRQ domain** 198
 - IRQ number** 198
 - irq_chip structure** 195-196
 - irq_create_mapping() function** 199-200
 - irq_data structure** 197-198
 - irq_desc structure** 196-197
 - irq_domain structure** 198
 - irq_domain_add_*(*) functions** 198
 - irq_set_chip_and_handler() function** 200
- ## J
- jiffie** 218
- ## K
- kernel memory, allocators**
 - kmalloc allocator** 343
 - PAGE allocator** 338
 - PAGE allocator API** 338-339
 - SLAB allocator** 339-341
 - SLAB allocator API** 342-343
 - kernel modules**
 - building and installing, on Raspberry Pi** 38

kernel object (kobject)

- attributes 50-52
- structures 49-51

kernel physical memory

- memory-mapped I/O 113, 336
- ZONE_DMA 336
- ZONE_HIGMEM 336
- ZONE_NORMAL 336

kernel threads

- definition 239
- kthread_create() function 239-240
- kthread_run() function 240
- kthread_stop() function 240
- wake_up_process() function 240

kfree() function 343**kmalloc allocator**

- about 343
- devm_kmalloc() function 343
- devm_kzalloc() function 139, 170, 182, 244, 378
- kfree() function 343
- kmalloc() function 343
- kzalloc() function 343

kthread_create() function 239-240**kthread_run() function 240****kthread_stop() function 240****L****led_classdev structure 134-135****Linux, address types**

- bus addresses 334
- kernel logical addresses 334
- kernel virtual addresses 335
- physical addresses 334
- user virtual addresses 333

Linux, boot process

- DDR initialization 28
- main stages 28-29
- on-chip boot ROM 28

start_kernel() function 29**Linux, device and driver model**

- about 41
- binding 44, 91, 167, 380, 414
- bus controller drivers 44
- bus core drivers 42-44
- bus_register() function 42-43, 163, 413
- bus_type structure 41, 42-45, 163, 413
- data structures 73
- device drivers 44-45
- device structure 46-47
- device_driver structure 45
- device_register() function 43
- driver_register() function 43, 44
- probe() function 44, 89, 90

Linux kernel

- about 22
- building, on Raspberry Pi 37-38
- distribution kernels 24
- licensing 62
- longterm 24
- mainline 23
- prepatch 23
- stable 23

Linux LED class

- about 133-134
- devm_led_classdev_register() function 134, 139, 178, 182

list_head devres_head structure 114**locking, kernel**

- about 226
- mutex 226
- spinlock 226

M**menuconfig 37****miscellaneous devices 85, 86****miscdevice structure 85**

`misc_deregister()` **function** 85
`misc_register()` **function** 85, 86
`mmap()` **function** 148, 151, 152, 335

MMIO (Memory-Mapped I/O)

about 113
`devm_ioremap()` **function** 114-115, 132, 136, 139, 154
`ioremap()` **function** 114
reading/writing, interfaces 115

MMU (Memory Management Unit) 333

MMU translation tables 333

`module_exit()` **function** 61
`module_init()` **function** 61
MODULE_LICENSE **macro** 62
`module_platform_driver()` **function** 90
`module_spi_driver()` **function** 416
multiplexing, pin 96
mutex, kernel lock 226

O

`of_device_id` **structure** 91
`of_match_table`
 I2C device driver 166
 platform device driver 89
 SPI device driver 417
`of_platform_populate()` **function** 58, 164, 167, 413
`of_property_read_string()` **function** 123
`of_scan_flat_dt()` **function** 56-57
`open()` **system call** 70

P

pad
 definition 96
PAGE allocator, kernel memory 338
PAGE allocator API, kernel memory 338-339
pin configuration node, Device Tree 295-299
pin control subsystem, about 97

pin, definition 96
`pinconf_ops` **structure** 101-102
`pinctrl_desc` **structure** 99, 100-103
`pinctrl_enable()` **function** 105
`pinctrl_init_controller()` **function** 105
`pinctrl_ops` **structure** 100-101
`pinctrl_register()` **function** 104
`pinmux_ops` **structure** 102-103
platform bus, about 42, 89
platform device driver
 about 89
 `module_platform_driver()` **function** 90
 platform_driver **structure** 89-90
 `platform_driver_register()` **function** 90
 `platform_get_irq()` **function** 133
 `platform_get_resource()` **function** 132-133
 `platform_set_drvdata()` **function** 124
 `probe()` **function** 44, 89, 90
 registering 89-90
 resources, structures and APIs 132-133
 `resource_size()` **function** 133
`platform_driver_register()` **function** 90
`platform_get_drvdata()` **function** 124
`platform_get_irq()` **function** 133
`platform_get_resource()` **function** 132-133
`platform_set_drvdata()` **function** 124
`probe()` **function** 44, 89, 90
process context, kernel 214, 221, 222, 227, 239
`put_user()` **function** 113

Q

`queue_work()` **function** 225

R

`read()` **system call** 70, 71
`register_chrdev_region()` **function** 72-73
Regmap

about 531

devm_regmap_init_i2c() function 532

devm_regmap_init_spi() function 532

regmap structure 531

regmap_bus structure 531

regmap_config structure 531, 534-535

Regmap, implementation

APIs 532-533, 535-536

regmap_config structure 543-544

regmap_read() function 533, 536

regmap_update_bits() function 533

regmap_write() function 533, 535

regmap_bulk_read() function 540

request_irq() function 206

resource structure 132

resource_size() function 133

root filesystem 27-28

S

schedule_delayed_work_on() function 224

schedule_on_each_cpu() function 224

schedule_work() function 224

setup_arch() function 54

setup_machine_fdt() function 54, 55, 56

set_bit() function 375, 379, 390-391

SLAB allocator, kernel memory 339-341

SLAB allocator API, kernel memory 342-343

sleeping, kernel

about 228-229

DECLARE_WAIT_QUEUE_HEAD() function 228

init_waitqueue_head() function 228, 232

wait queue 228

wait_event() function 228-229

wait_event_interruptible() function 229

wait_queue_head_t structure 229, 232

wake_up() function 229

SMBus, definition 161

softirqs, deferred work 215-216

SPI, about 411

SPI device drivers

about 415-416

registration 416-417

spi_driver structure 416

SPI, Linux

bus core drivers 413

controller drivers 413-415

protocol drivers 415-416

spi_async() function 412, 413

spi_read() function 412

spi_sync() function 412

spi_write() function 412

spi_write_then_read() function 412, 415, 429

spinlock, kernel lock 226

spi_w8r16() function 412

spi_write_then_read() function 412, 415, 429

start_kernel() function 29

sysfs filesystem

about 47-48

system call, interface

about 25

open() system call 70

read() system call 70

write() system call 70

system shared libraries

about 26

locations 27

T

tasklets, deferred work 217

threaded interrupts, deferred work 221-222

timers, deferred work 217-218

timer_list structure 217

toolchain

about 29

setting up, on Raspberry Pi 37

U

UIO framework

- APIs 151-152

- definition 148-150

- working 150-151

`uio_register_device()` function 152

`unflatten_device_tree()` function 58

Unified device properties, API

- about 180

- functions 180

`unregister_chrdev_region()` function 72

USB, about 563

USB descriptors

- about 569

- USB configuration descriptor 571-572

- USB device descriptors 569-571

- USB endpoint descriptor 573

- USB HID descriptor 574-576

- USB interface descriptor 572-573

- USB string descriptor 573-574

USB device driver

- completion handler 581, 583-584, 600

- registering 578-579

- `urb` structure 582

- `usb_alloc_urb()` function 582

- `usb_endpoint_descriptor` structure 580-581

- `usb_free_urb()` function 583

- `usb_host_endpoint` structure 580

- `usb_interface` structure 579-580

- `usb_kill_urb()` function 583-584

- `usb_submit_urb()` function 581, 583

- `usb_unlink_urb()` function 581, 583-584

USB request block (URB) 581-584

USB subsystem, Linux

- USB bus core drivers 576-577

- USB device drivers 577-579

- USB host-side drivers 579-581

User space, drivers

- advantages 147

- disadvantages 147

U-Boot

- about 20

- main features 20-21

V

virtual file, about 65, 69

virtual interrupt ID 195

virtual memory layout, user space process

- data segment 335

- memory mapping segment 335

- stack segment 335

- text segment 335

virtual to physical, memory mapping, kernel
336-337

W

wait queue, kernel sleeping 228

`wait_event()` function 228-229

`wait_event_interruptible()` function 229

`wake_up()` function 229

workqueues, deferred work

- about 223-226

- `create_singlethread_workqueue()` function 225

- `create_workqueue()` function 225-226

- `DECLARE_WORK()` function 224

- `destroy_workqueue()` function 225

- `flush_scheduled_work()` function 224

- `flush_workqueue()` function 225

- `INIT_WORK()` function 224

- `queue_work()` function 225

- `schedule_delayed_work_on()` function 224

- `schedule_on_each_cpu()` function 224

- `schedule_work()` function 224

- work item 223

- worker 223

workqueue_struct **structure** 225
work_struct **structure** 223
write() **system call** 70, 71

